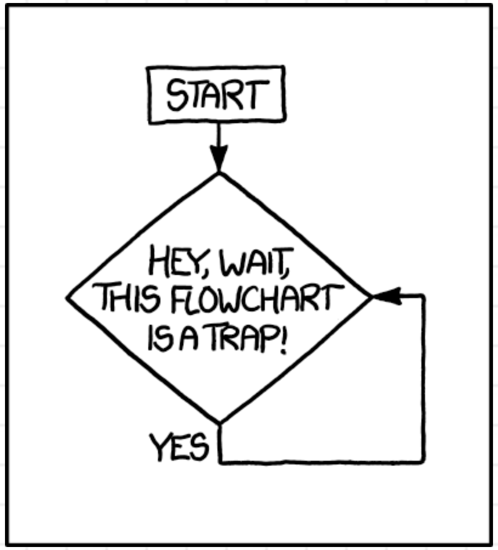
CS-3510
Design and Analysis of Algorithms

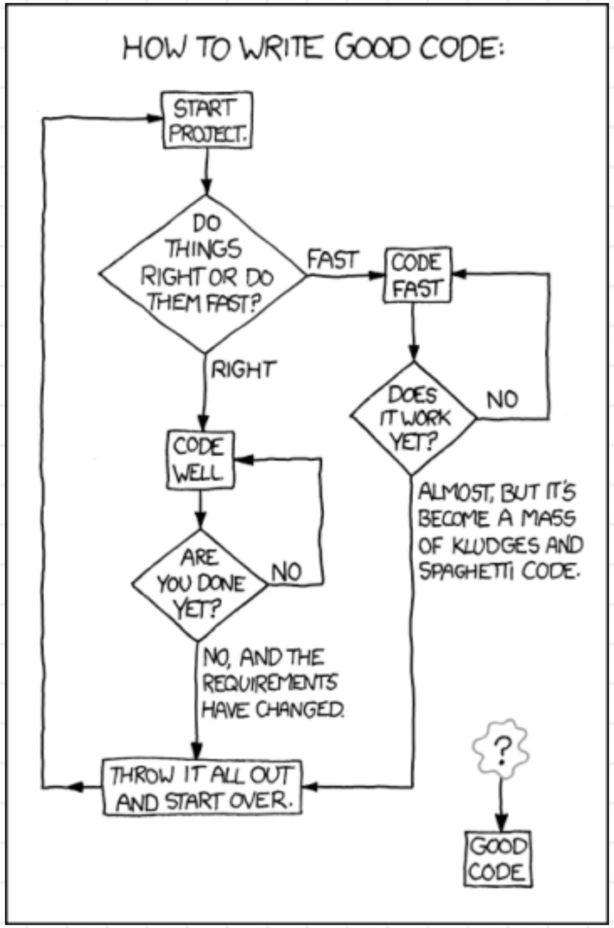
Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Welcome!

- Course: <u>Design and Analysis of Algorithms</u>
- What?
  - Algorithms
  - Algorithmic paradigms; design and correctness
  - Performance analysis





Welcome!

- Why?
  - Fundamental to all areas of computer science
    - Operating systems, Networks and distributed systems, Machine learning, Data science, Numerical computation, Cryptography, Computational biology, etc.
  - Inseparable part of every technical interview | *We talk about this more!*
    - Internship, Part-time, Full-time
    - Software engineer (SWE), Machine learning engineer (MLE), Product manager (PM), Infrastructure, Research scientist, Data scientist, etc.
    - Large tech companies ~ small start-ups
  - Useful and Fun!
    - Problem solving skills
    - Competitive programming, Hackathons, etc.

Welcome!

- When?
  - Days: Tuesday Thursday
  - Time: 3:30 - 5:40 pm EST

- Where?
  - Klaus Advanced Computing 2443

- Prerequisites?
  - (Some) discrete math and data structure knowledge

1. CS 2050 or CS 2051 or MATH 2106
2. CS 1332 or MATH 3012 or MATH 3022

Logistics

- How? | Course Format
  - In-person lectures
  - In-person exams

- Lectures will NOT be recorded!
  - CoC does not provide recording option.
  - Remote/virtual options are not available.

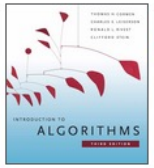
- How to access the course material?
  - Course website: http://www.cs3510.com/

Logistics

- Course Website: http://www.cs3510.com/
  - Course materials
    - General schedule
    - Notes, slides, demo codes
    - Textbooks
    - Assignments
    - Exam dates
  - Policies
    - Grading
    - Homework assignments
    - Late policies, regrade policies
    - Collaboration and honor code
  - Check **regularly** for announcements and course materials.

Logistics

- Course Website: http://www.cs3510.com/
  - Course materials
    - General schedule http://www.cs3510.com/lectures/
    - Notes, slides, demo codes
    - Textbooks http://www.cs3510.com/policies/#2-textbooks/
    - Assignments http://www.cs3510.com/assignments/
    - Exam dates
  - Policies
    - Grading
    - Homework assignments
    - Late policies, regrade policies
    - Collaboration and honor code



Required

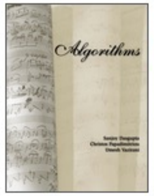
CLRS

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. (Full text is available online for Georgia Tech students)

Recommended | Optional

KT

Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.



Recommended | Optional

DPV

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.

- Check **regularly** for announcements and course materials.

Logistics

- Course Website: http://www.cs3510.com/
  - Course materials
    - General schedule
    - Notes, slides, demo codes
    - Textbooks
    - Assignments
    - Exam dates
  - Policies
    - Grading
    - Homework assignments
    - Late policies, regrade policies
    - Collaboration and honor code

Scheme 1:	Scheme 2:
- Homeworks: 30%	- Homeworks: 30%
- Exam 1: 30%	- Exam 1: 15%
- Exam 2: 30%	- Exam 2: 15%
- Final Exam: 10%	- Final Exam: 40%

3.2. Letter Grade Cutoff

- A 90-100%
- B 80-90%
- C 70-80%
- D 60-70%
- F 0-60%

- Check **regularly** for announcements and course materials.

Logistics

- Course Website: http://www.cs3510.com/

  - Course materials
    - General schedule
    - Notes, slides, demo codes
    - Textbooks
    - Assignments
    - Exam dates

  - Policies
    - Grading
    - Homework assignments
    - Late policies, regrade policies
    - Collaboration and honor code

- 6-7 assignments
- Every Friday, due the next Friday 11:59 pm EST
- First assignment will be released this Friday!
- Solutions must be typed
  - LaTeX is highly recommended
  - Tex template file will be provided
  - A cloud-based LaTeX editor: OverLeaf (free for GT)

- Check **regularly** for announcements and course materials.

Logistics

- Course Website: http://www.cs3510.com/

  - Course materials
    - General schedule
    - Notes, slides, demo codes
    - Textbooks
    - Assignments
    - Exam dates

  - Policies
    - Grading
    - Homework assignments
    - Late policies, regrade policies
    - Collaboration and honor code

- 6 total late days for the entire semester with no penalty
- At most 2 late days can be used for one assignment
- The late days are counted by day; a new late day starts at 12:00 am EST
- Submissions beyond the total allowed late days or 2 days after the deadline will get 0 credit.
- Regrade request
  - Within one week from the day grades published
  - Directly email to the corresponding TA

- Check **regularly** for announcements and course materials.

Logistics

- Course Website: http://www.cs3510.com/
  - Course materials
    - General schedule
    - Notes, slides, demo codes
    - Textbooks
    - Assignments
    - Exam dates
  - Policies
    - Grading
    - Homework assignments
    - Late policies, regrade policies
    - Collaboration and honor code

- Acknowledge the code of academic integrity
- Collaboration is allowed but
  - You must write up and submit your own work
  - You must acknowledge (explicitly mention) your collaborators
- Proper citation is required for any material used outside the lectures.

- Check **regularly** for announcements and course materials.

Logistics

- So, in short:

  - All course materials: http://www.cs3510.com/
    (lecture notes, assignments, solutions, policies, etc.)

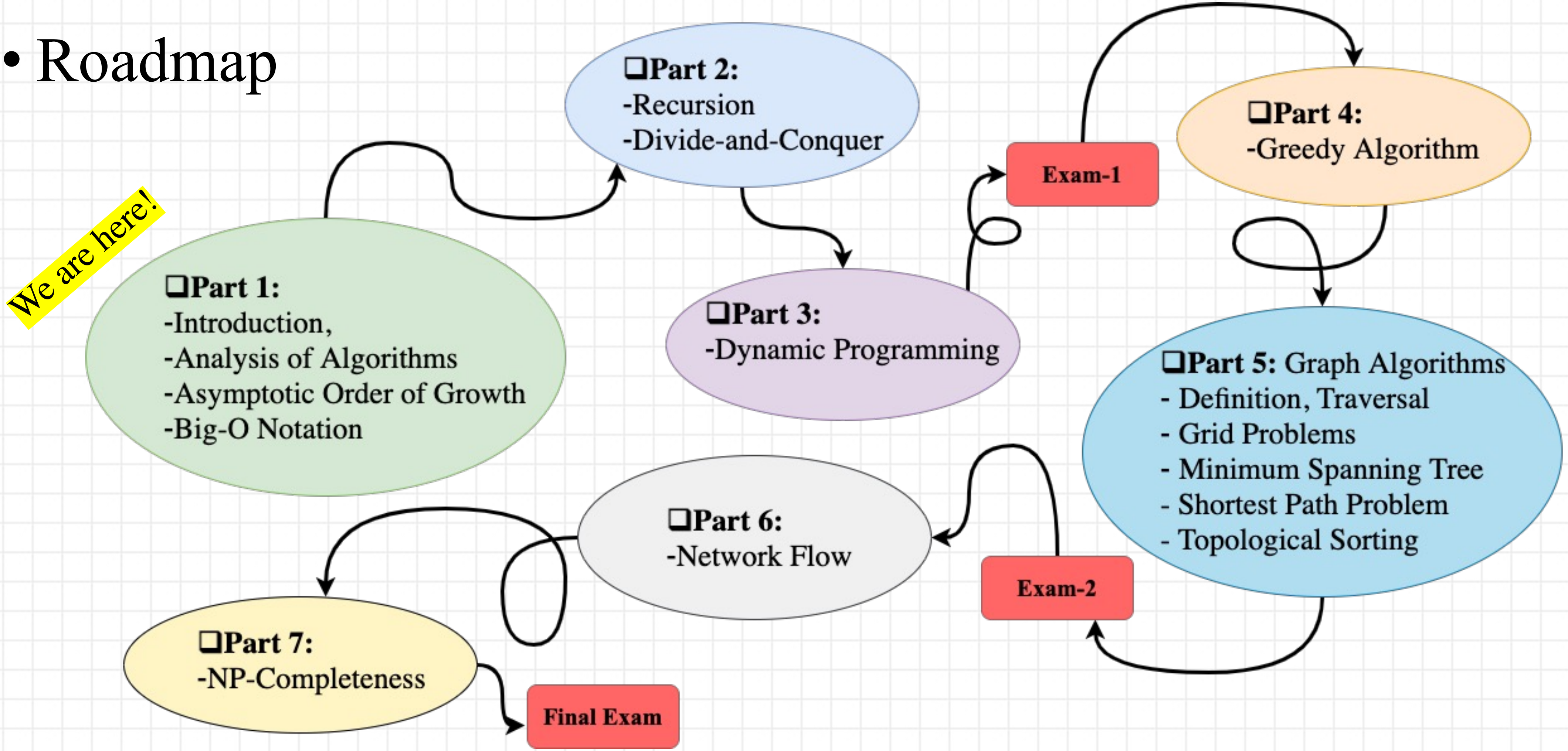
  - Assignment submission: Canvas

Logistics

- Communications:
  - Discussions: Piazza
    - Join today!
    - Class Access Code: **cs3510a--summer2022**
  - Email and Office hours
    - Office hours start from the next week
    - Information in the course website
    - Locations will be announced; most probably online Zoom meeting

Course Plan

- Roadmap



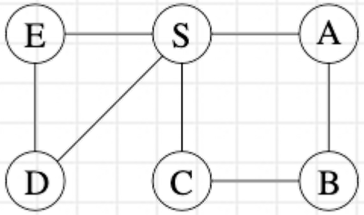
Course Content

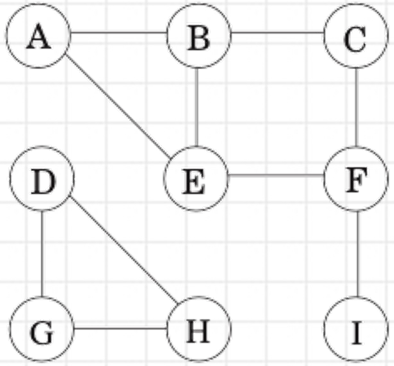
- Plan1: More theoretical
  - More mathematical proof
  - More classic textbook problems in examples and assignments

- Plan 2: More practical
  - Technical interview problems in examples and assignments
    - Internship/fulltime
  - More demo codes

Course Content

- An example:
- Textbook: Graph Traversal Problem
  - Breadth-First/Depth-First Search
  - Design an algorithm to verify if there is a path between two nodes in a given graph.





- Interview: Grid Problem
  - Given an m-by-n 2D binary matrix in which 0 represent water and 1 represent land, design an algorithm computing the number islands. An island includes one or more horizontally or vertically cells surrounded by water.

1	1	0	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	0	0

1	1	0	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	0	0

Course Content

- Plan1: More theoretical
  - More mathematical proof
  - More classic textbook problems in examples and assignments
- Plan 2: More practical
  - Technical interview problems in examples and assignments
  - More demo codes
- Mentimeter
  - Use your smart phone or laptop to vote:
    - https://www.menti.com/
    - Code: 37 01 46
  - We will use Mentimeter for in-class quizzes, as well!

Algorithm Analysis

- Algorithm (Meriam-Webster Dictionary):
  - "A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation"
  - "Broadly **:** a step-by-step procedure for solving a problem or accomplishing some end."
- Algorithm (CLRS):
  - "Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output."
  - "An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem."

Algorithm Analysis

- Correctness
  - On every valid input, the algorithm produces the output that satisfies the required input/output relationship
- Proof of Correctness
  - Induction
  - Contradiction
  - Counter-example
  - …
- But we are also interested beyond correctness!
  - The resources that an algorithm uses: time and space

Algorithm Analysis

- Analysis of Time and Space
  - Space: main memory used by an algorithm
  - Time: the number of CPU cycles used
  - We commonly are most interested in time (speed)
  - Space-time trade off
    - Dynamic programming
  - Here, we will review the time complexity, but the same discussion holds for space complexity.

  - Given the size of the input, how many computation steps does the algorithm take?
    - Running time
    - Time complexity

Running time (review)

- Model of Computation:
  - Generic one-processor, random-access machine (RAM)
  - No concurrency: instructions are executed one after another
  - Common atomic instructions: take constant amount of time
    - Arithmetic (e.g., add, subtract, multiply, divide, remainder, floor, ceiling)
    - Data movement (load, store, copy)
    - Control (conditional and unconditional branch, subroutine call and return)
  - Loops are not simple operations
    - Depends on the size of the input
  - Note call a subroutine takes constant amount of time but the subroutine runtime may not be constant.
- Runtime: number of steps taken by an algorithm, given the input size

Running time (review)

- Best-case: the minimum number of steps taken on any instance of size n
  - Example: sorting an array when the input array is already sorted

- Worse-case: the maximum number of steps taken on any instance of size n
  - Example: sorting an array when the input array is reverse sorted

- Average-case: the average number of steps taken on any instance of size n

- Time complexity:
  - The time complexity of an algorithms associates a number T(n) defined as the maximum amount of time, i.e., the worst case, taken on any input of size n.

Time Complexity (review)

- Time complexity:
  - The time complexity of an algorithms associates a number T(n) defined as the maximum amount of time, i.e., the worst case, taken on any input of size n.

  - Mathematical definition:
    - $T(n)\colon \mathbb{N} \rightarrow \mathbb{R}$
    - T(n) represents a mapping from input size n, which is a non-negative integer, to a real number showing the runtime in the worst-case scenario for any input of size n.

  - Problem/Limitation:
    - The exact analysis is often hard due to implementation details, etc.
    - How the runtime is scaling-up if the input size increases? Rate of growth

  - Better approach: asymptotic analysis

Time Complexity (review)

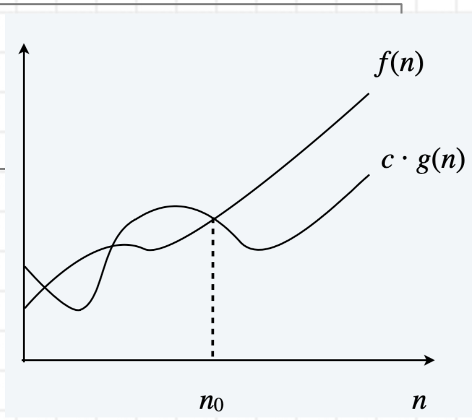
- Asymptotic Order of Growth
  - It is easier to talk about the lower bound and upper bound of the running time.
  - To practically deal with time complexity analysis, we use asymptotic notations.
  - The asymptotic growth of a function (in this case $T(n)$) is specified using Θ, O, and Ω notations.
  - Asymptotic means for "very large" input size, as n grows without bound or "asymptotically".

Time Complexity (review)

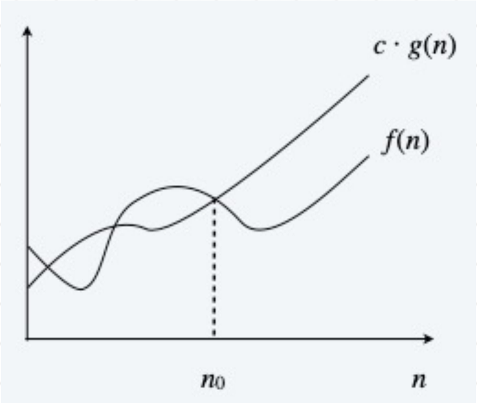
- Asymptotic Order of Growth
  - In general, the asymptotic notations define bounds on the growth of a function. Informally, a function $f(n)$ is:
    - $\Omega(g(n))$ if $g(n)$ is an asymptotic lower bound for $f(n)$
    - $O(g(n))$ if $g(n)$ is an asymptotic upper bound for $f(n)$
    - $\Theta(g(n))$ if $g(n)$ is an asymptotic tight bound for $f(n)$

Time Complexity (review)

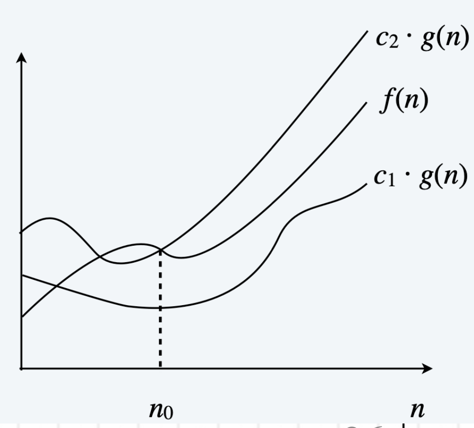
- Asymptotic Order of Growth (Formal definition):
  - **Big Omega (lower bound):**
    $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq cg(n) \geq 0$ for all $n \geq n_0$.



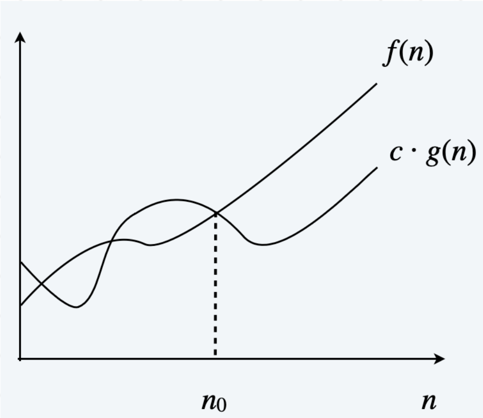
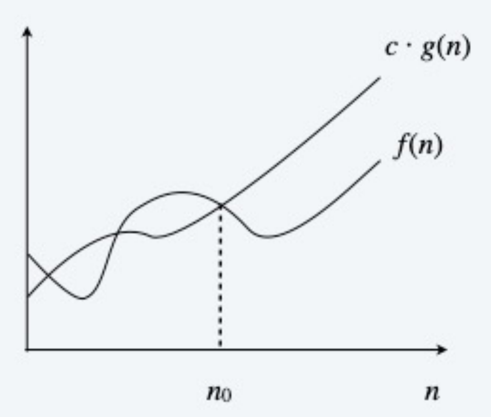
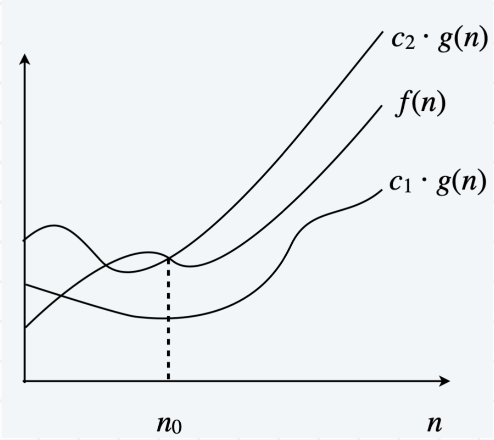
  - **Big O (upper bound):**
    $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$



  - **Big Theta (tight bound):**
    $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
  - Note: $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

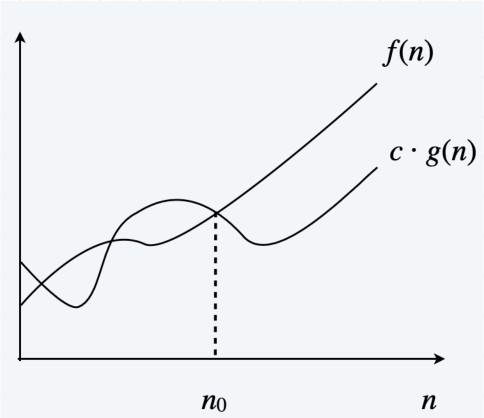
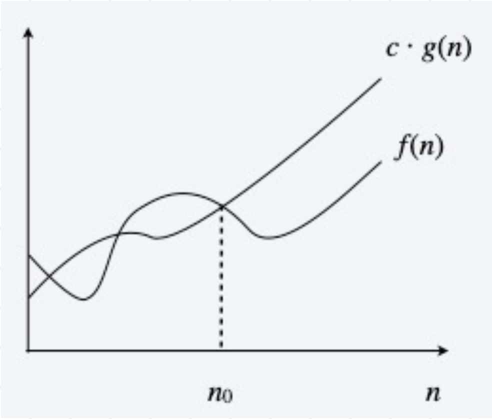
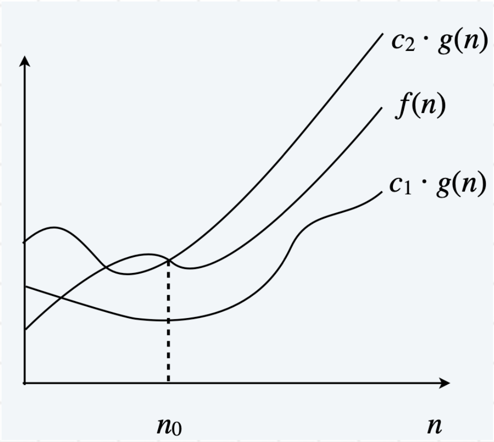


Time Complexity (review)

		
$f(n)$ is $\Omega(g(n))$ $g(n)$ is a lower bound of $f(n)$	$f(n)$ is $O(g(n))$ $g(n)$ is an upper bound of $f(n)$	$f(n)$ is $\Theta(g(n))$ $g(n)$ is a tight bound of $f(n)$

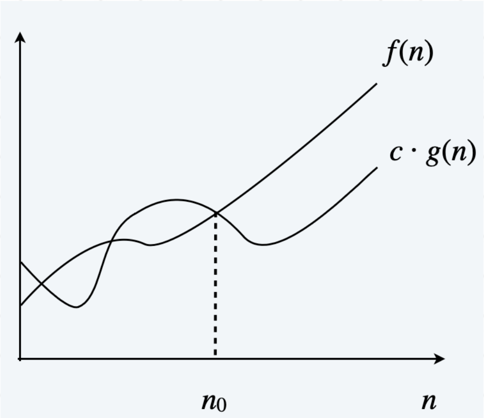
- Ex. Let $f(n) = 35n^2 + 10n + 5$. Then, we can say:
  - $f(n)$ is $O(n^2), O(n^3), \Omega(n), \Omega(n^2)$, and $\Theta(n^2)$.
  - $f(n)$ is not $O(n), O(n\log n), \Omega(n^3), \Theta(n), \Theta(n^3)$

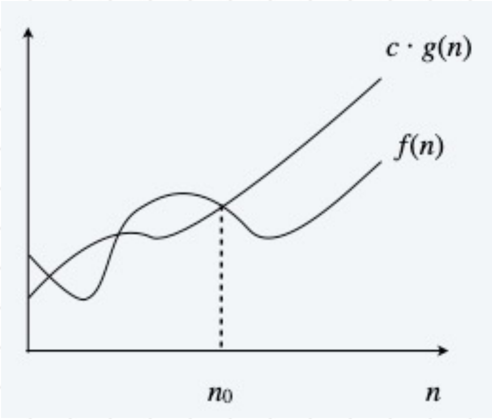
Time Complexity (review)

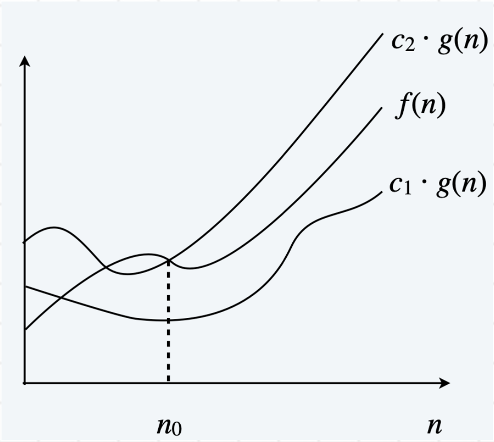
		
$f(n)$ is $\Omega(g(n))$ $g(n)$ is a lower bound of $f(n)$	$f(n)$ is $O(g(n))$ $g(n)$ is an upper bound of $f(n)$	$f(n)$ is $\Theta(g(n))$ $g(n)$ is a tight bound of $f(n)$

- Ex. Let $f(n) = 35n^2 + 10n + 5$. Then, we can say:
  - $f(n) \in O(n^2), O(n^3), \Omega(n), \Omega(n^2)$, and $\Theta(n^2)$.
  - $f(n) \notin O(n), O(n\log n), \Omega(n^3), \Theta(n), \Theta(n^3)$.
- $f(n) \in O\big(g(n)\big)$ means $f(n)$ is in the set of functions bounded by $g(n)$ from above

Time Complexity (review)



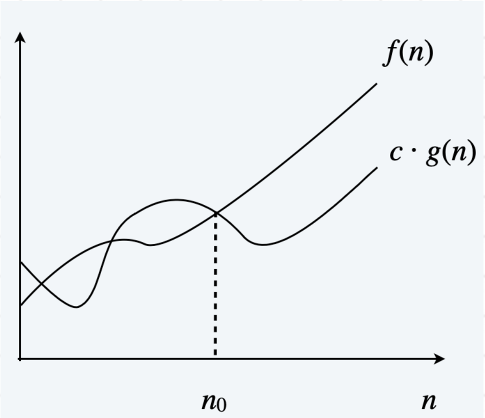
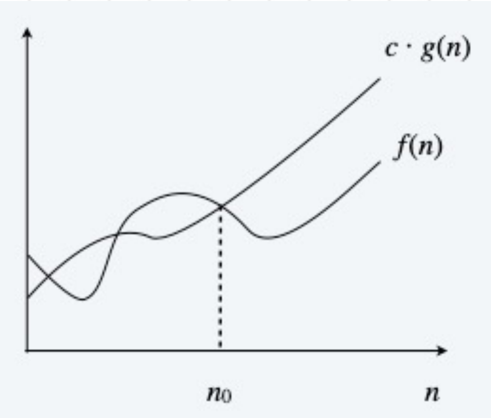
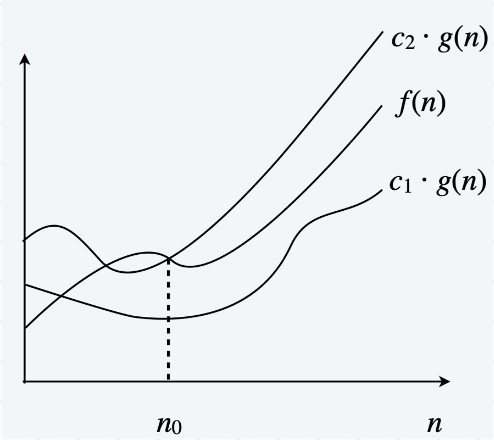




$f(n)$ is $\Omega(g(n))$ $g(n)$ is a lower bound of $f(n)$	$f(n)$ is $O(g(n))$ $g(n)$ is an upper bound of $f(n)$	$f(n)$ is $\Theta(g(n))$ $g(n)$ is a tight bound of $f(n)$

- $f(n) \in O\big(g(n)\big)$ means $f(n)$ is in the set of functions bounded by $g(n)$ from above
- Slight abuse of asymptotic notations: $f(n) = O\big(g(n)\big)$
  - Often used by computer scientist
  - Problem: equality is not transitive. (Ex. Let $g_1(n) = 7n^3$ and $g_2(n) = 2n^3$. We have $g_1(n) = O(n^3)$ and $g_2(n) = O(n^3)$, but we cannot conclude $g_1(n) = g_2(n)$)

Time Complexity (review)

		
$f(n)$ is $\Omega(g(n))$ $g(n)$ is a lower bound of $f(n)$ $f(n) \in \Omega(g(n))$	$f(n)$ is $O(g(n))$ $g(n)$ is an upper bound of $f(n)$ $f(n) \in O(g(n))$	$f(n)$ is $\Theta(g(n))$ $g(n)$ is a tight bound of $f(n)$ $f(n) \in \Theta(g(n))$

- The time complexity of an algorithms associates a number T(n) defined as the maximum amount of time, i.e., the worst case, taken on any input of size n.
- Big-O: $\boxed{T(n) \in O(g(n))}$

Time Complexity (review)

- Big O Notation Properties

Reflexivity	f is $O(f)$
Constants	If f is $O(g)$ and $c > 0$, then cf is $O(g)$
Products	If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$
Sums (Additivity)	If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max \{g_1, g_2\})$ Ex. If $f_1 \in O(n^2)$ and $f_2 \in O(n^4)$. Then, $f_1 + f_2 \in O(n^4)$
Transitivity	If f is $O(g)$ and g is $O(h)$, then f is $O(h)$

- So, we can ignore the lower terms and constants:
  - Ex. $f = 2n^3 + 4n^2 - 5n + 1 \in O(n^3)$
  - Ex. $f = 4n^5 \in O(n^5)$

Time Complexity (review)

- Asymptotic Bounds for Some Common Functions

Polynomials	$f(n) = a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ and thus, $O(n^d)$ if $a_d > 0$.
Logarithms	$\log_a n$ is $\Theta(\log_b n)$ for every $a>1$ and $b>1$. Note: $O(\log_a n) = O(\log_b n)$ (Recall $\log_b n = \log_b a \times \log_a n$)
Logarithms vs polynomials	$\log_a n$ is $O(n^d)$ for every $a>1$ and $d>0$. Logarithms grow slower than every polynomial regardless of how small d is.
Exponential vs Polynomials	n^d is $O(r^n)$ for every $d>0$ and $r>1$. Exponentials grow faster than every polynomial regardless of how big d is.

- Demo code

Common Running Times

(1) Constant time: Running time is $O(1)$

- Bounded by a constant which does not depend on input size n.

- Examples
  - Arithmetic/logic operation
  - Declare/initialize a variable
  - Access an element in an array
  - Follow a link in a linked list
  - Conditional branch

Common Running Times

(2) Linear time: Running time is $O(n)$

- The runtime scales up linearly with respect to the input size n.

- Examples
  - Finding the minimum and maximum elements in an array or linked list
  - Searching for an element in an unsorted array
  - Combining two sorted list

Common Running Times

(3) Logarithmic time: Running time is $O(\log n)$

- The runtime scales up logarithmically with respect to the input size n.

- Examples
  - Binary Search in a sorted array.
  - Finding the target sum in a sorted array.

(4) $O(n\log n)$:

- Sorting elements of an array in ascending order using Merge-Sort algorithm

Common Running Times

(5) $O(n^2)$

- Algorithm to solve the closest pair of points problem. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest to each other.

(6) $O(n^3)$

- Given an array of n distinct integers, find three that sum to 0.

(7) Polynomial time: Running time is $O(n^k)$ for some constant $k > 0$.

- Independent set of size k. Given a graph, find k nodes such that no two are joined by an edge.
- In general, an algorithm is considered efficient if it has polynomial running time.

Common Running Times

(8) Exponential time: Running time is $O\left(2^{n^k}\right)$ for some constant $k > 0$.

- $O(2^n)$: Enumerating all subsets of a set of n elements.

References

- The lecture slides are heavily based on the suggested textbooks and the corresponding published lecture notes:

  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
  - DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.

CS-3510: Design and Analysis of Algorithms

Some Examples

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Searching a Sorted Array

- Problem
  - Given a sorted array, including integer numbers, and a target number, design an algorithm which returns True if the target number is in the given array, and False otherwise.
  - Input: $A = [a_1, a_2, \ldots, a_N]$ s.t. $a_1 < a_2 < \ldots < a_N$, target = k
  - Output: True if $k \in A$, False if $k \notin A$

- Example
  - Input: A = [-1, 0, 2, 5, 8, 11], target = 8
  - Output: True

Searching a Sorted Array

- Approach 1:
  - Brute force search: checking every element one-by-one
  - In this case: Linear Search

- A = [-1, 0, 2, 5, 8, 11], target = 8
- A = [-1, 0, 2, 5, 8, 11], target = 8
- A = [-1, 0, 2, 5, 8, 11], target = 8
- A = [-1, 0, 2, 5, 8, 11], target = 8
- A = [-1, 0, 2, 5, 8, 11], target = 8
- A = [-1, 0, 2, 5, 8, 11], target = 8

Return True!

Searching a Sorted Array

- Approach 1:
  - Brute force search: checking every element one-by-one
  - In this case: Linear Search
  - Time complexity: O(n)
  - Space complexity: O(1)

Algorithm 1: `Linear Search`

Input: $A = \{a_1, a_2, \ldots, a_n\}, k$
Result: $True$ or $False$

```
for (i = 1 to n) do
    if k == a_i) then
        // the target is in the array
        return True
end
return False
```

- What if n=1,000,000 and k == a_n?
- Can we do better?

Searching a Sorted Array

- Binary Search
  - Assume the value of the mid element is less than the target, do we still need search the left half?
  - At each iteration compare the mid element of with the target:
    - if mid == target: return True
    - if mid < target: discard the left sub-array and continue to search the right sub-array
    - if mid > target: discard the right sub-array and continue to search the left sub-array

  - Naturally can be implemented as a recursive algorithm.
  - Recursion
    - Function call itself on a smaller domain
    - One or more base cases to stop the recursion

Searching a Sorted Array

- Approach 2: Binary Search

lo hi

- A = [-1, 0, 2, 5, 8, 11], target = 8

lo m hi

- A = [-1, 0, **2**, 5, 8, 11], target = 8

2 < 8 → search right sub-array

lo hi

- A = [~~-1, 0, 2~~, 5, 8, 11], target = 8

lo m hi

- A = [~~-1, 0, 2~~, 5, **8**, 11], target = 8

- Return True!

Searching a Sorted Array

- Binary Search

Algorithm 3: `Binary Search (Recursive)`

Input: $A = \{a_1, a_2, ..., a_n\}, k, lo = 1, hi = n$
Result: $True$ or $False$

`// base case`
if $(lo > hi)$ **then**
    return $False$
end

`// recurrence relation`
$mid \leftarrow [(lo + hi)/2]$
if $(a_{mid} == k)$ **then**
    `// the target is in the array`
    return $True$
else if $(a_{mid} < k)$ **then**
    return `BinarySearch(A, k, mid+1, hi)`
else
    return `BinarySearch(A, k, lo, mid-1)`
end

Algorithm 2: `Binary Search (Iterative)`

Input: $A = \{a_1, a_2, ..., a_n\}, k$
Result: $True$ or $False$

$lo \leftarrow 1$
$hi \leftarrow n$
while $(lo \le hi)$ **do**
    $mid \leftarrow [(lo + hi)/2]$
    if $(a_{mid} == k)$ **then**
        `// the target is in the array`
        return $True$
    else if $(a_{mid} < k)$ **then**
        $lo \leftarrow mid + 1$
    else
        $hi \leftarrow mid - 1$
    end
end

return $False$

Searching a Sorted Array

- Binary Search
  - Time complexity?
  - Recursive Algorithms
    - Recursion tree
    - Substitution | Guess and prove by induction
    - Master theorem

Algorithm 3: `Binary Search (Recursive)`

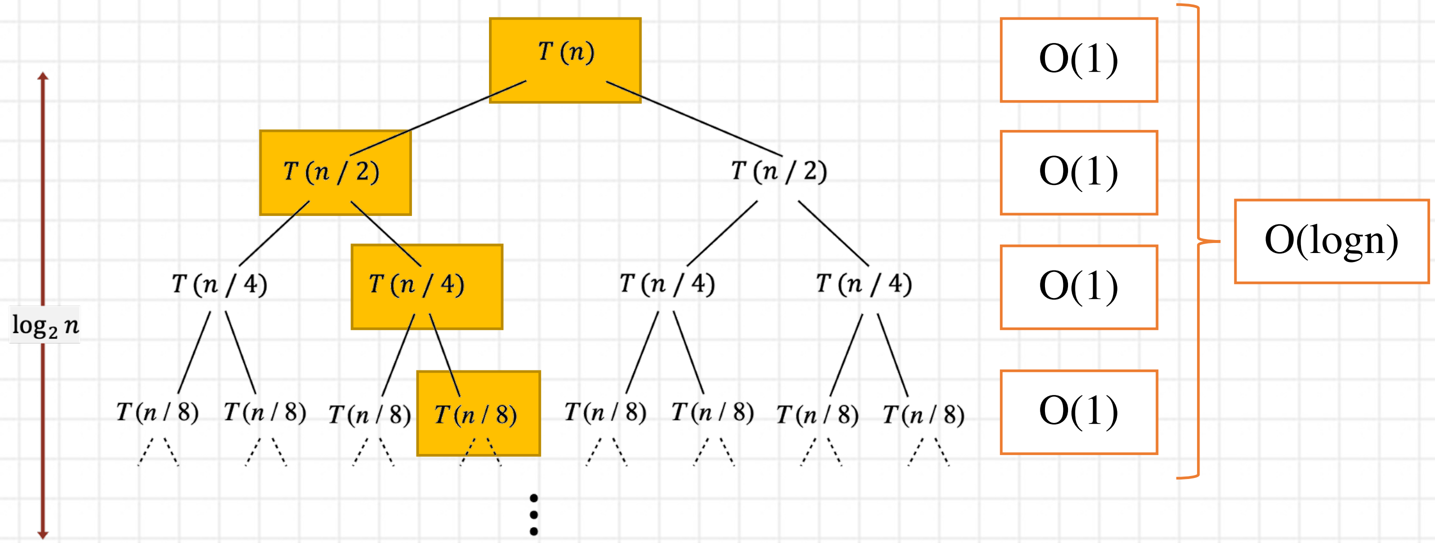
Input: $A = \{a_1, a_2, ..., a_n\}, k, lo = 1, hi = n$
Result: $True$ or $False$

```
// base case
if (lo > hi) then
    return False
end

// recurrence relation
mid ← [(lo + hi)/2]
if (a_mid == k) then
    // the target is in the array
    return True
else if (a_mid < k) then
    return BinarySearch(A, k, mid+1, hi)
else
    return BinarySearch(A, k, lo, mid-1)
end
```

Searching a Sorted Array

- Binary Search
  - Time complexity?
  - Recursive Algorithms
    - **Recursion tree**
    - Substitution | Guess and prove by induction
    - Master theorem



Algorithm 3: `Binary Search (Recursive)`

Input: $A = \{a_1, a_2, ..., a_n\}, k, lo = 1, hi = n$
Result: $True$ or $False$

```
// base case
if (lo > hi) then
    return False
end

// recurrence relation
mid ← ⌊(lo + hi)/2⌋
if (a_mid == k) then
    // the target is in the array
    return True
else if (a_mid < k) then
    return BinarySearch(A, k, mid+1, hi)
else
    return BinarySearch(A, k, lo, mid-1)
end
```

Searching a Sorted Array

- Binary Search
  - Time complexity?
  - Recursive Algorithms
    - Recursion tree
    - **Substitution** | Guess and prove by induction
    - Master theorem

    - mid is always placed at $1 + \lfloor n/2 \rfloor$
    - if $k < a_{mid}$ we have $\lfloor n/2 \rfloor$ to search
    - else:
      - $\lfloor n/2 \rfloor$ if n is odd
      - $\lfloor n/2 \rfloor - 1$ if n is even

Algorithm 3: `Binary Search (Recursive)`

Input: $A = \{a_1, a_2, \ldots, a_n\}, k, lo = 1, hi = n$
Result: $True$ or $False$

```
// base case
if (lo > hi) then
    return False
end

// recurrence relation
mid ← ⌊(lo + hi)/2⌋
if (a_mid == k) then
    // the target is in the array
    return True
else if (a_mid < k) then
    return BinarySearch(A, k, mid+1, hi)
else
    return BinarySearch(A, k, lo, mid-1)
end
```

Searching a Sorted Array

- Binary Search
  - Time complexity?
  - Recursive Algorithms
    - Recursion tree
    - **Substitution** | Guess and prove by induction
    - Master theorem

    - The recurrence: $T(n) = \begin{cases} 1 & , n = 1 \\ 1 + \lfloor n/2 \rfloor, & n > 1 \end{cases}$
    - Assume n is a power of 2: $\lfloor n/2 \rfloor = n/2$
    - Substitution, telescope the recurrent:

$$T(\mathrm{n}) = 1 + T\left(\frac{n}{2}\right) = 1 + 1 + T\left(\frac{n}{4}\right) = 1 + 1 + 1 + T\left(\frac{n}{8}\right) = \overbrace{1 + 1 + \ \ldots + 1}^{1+\log(n)} \in O(\log(n))$$

Algorithm 3: `Binary Search (Recursive)`

Input: $A = \{a_1, a_2, \ldots, a_n\}, k, lo = 1, hi = n$
Result: $True$ or $False$

```
// base case
if (lo > hi) then
    return False
end

// recurrence relation
mid ← ⌊(lo + hi)/2⌋
if (a_mid == k) then
    // the target is in the array
    return True
else if (a_mid < k) then
    return BinarySearch(A, k, mid+1, hi)
else
    return BinarySearch(A, k, lo, mid-1)
end
```

Searching a Sorted Array

- Binary Search
  - Time complexity?
  - Recursive Algorithms
    - Recursion tree
    - Substitution | Guess and prove by induction
    - Master theorem
      - Divide-and-Conquer

Algorithm 3: `Binary Search (Recursive)`

Input: $A = \{a_1, a_2, ..., a_n\}, k, lo = 1, hi = n$
Result: $True$ or $False$

`// base case`
if $(lo > hi)$ **then**
  return $False$
end

`// recurrence relation`
$mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
if $(a_{mid} == k)$ **then**
  `// the target is in the array`
  return $True$
else if $(a_{mid} < k)$ **then**
  return `BinarySearch(A, k, mid+1, hi)`
else
  return `BinarySearch(A, k, lo, mid-1)`
end

- Binary search is a divide-and-conquer approach
  - Divide up problems into several subproblems (of the same type)
  - Solve (conquer) each subproblem (usually recursively)
  - Combine the solutions

Searching a Sorted Array

- Comparing the running time
  - Demo code