DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

Course Instructor: Dr. Shreya Ghosh
TA(s)

INTRODUCTION (CI, TA(S) AND STUDENTS)

- **Btech** (IIEST, 2015), **Masters and PhD** (IIT Kharagpur, 2021), **Postdoctoral researcher** (Pennsylvania State University, USA, 2021-2023)
- **Research Interests**: **ML (NLP, Reinforcement Learning** and **Knowledge Graph), Spatio-temporal data analytics, Big data analytics**
- **Find more about my projects:**
  - https://scholar.google.co.in/citations?user=a5OKo7wAAAAJ&hl=en
  - https://shreghosh.github.io/

- **Research Collaboration(s) and Teaching**: ***University of Melbourne, Australia***, ***Foundation of Google Cloud computing*** (with Google Cloud research team), ***MBZUAI, Abu Dhabi, Penn State, USA.***

Reach out shreya@iitbbs.ac.in if you are interested in doing ML projects!

COURSE STRUCTURE

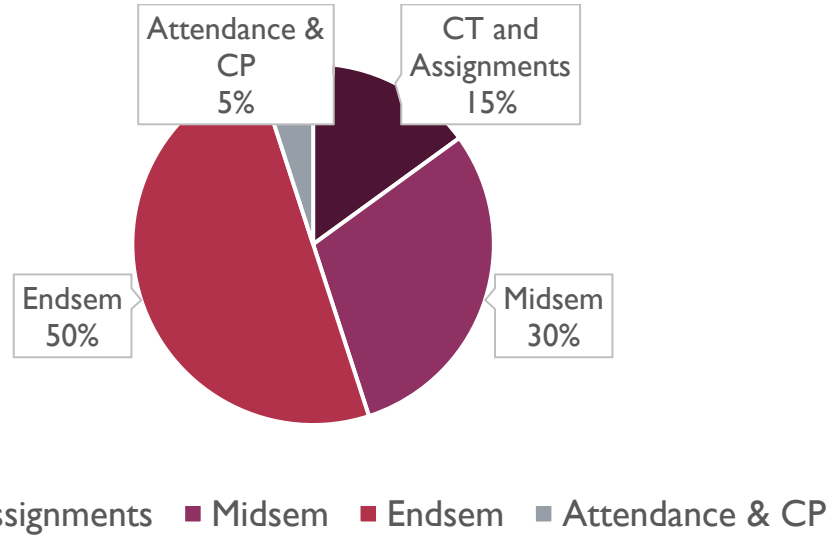
- **Design and analysis of algorithms (DAA)**
  - L-L-L-T [**Tue (14:30-16:30, SES-318)** and **Wed (16:30-18:30, LC11-120-1)]**
  - **Tutorial:** Solve and discuss Assignments/ New problems, Talk about recent Algorithm advances.

- Prerequisites: CS1L001 (Introduction to Programming and Data Structure), CS2L004 (Data Structures), Basics of Probability. [suggest to revise: *recursive functions, fundamental data structures, graphs, sets, functions, ands relations, random variables and their properties – linearity of expectation, the union bound, and conditional probabilities*]

- Website: https://sites.google.com/view/daa24/

COURSE STRUCTURE

- **Text books:**
  - Thomas H. Cormen, Charles E. Leiserson, R.L. Rivest. **Introduction to Algorithms**, Prentice Hall of India Publications.
  - J. Kleinberg and E. Tardos. **Algorithm Design**, Pearson.



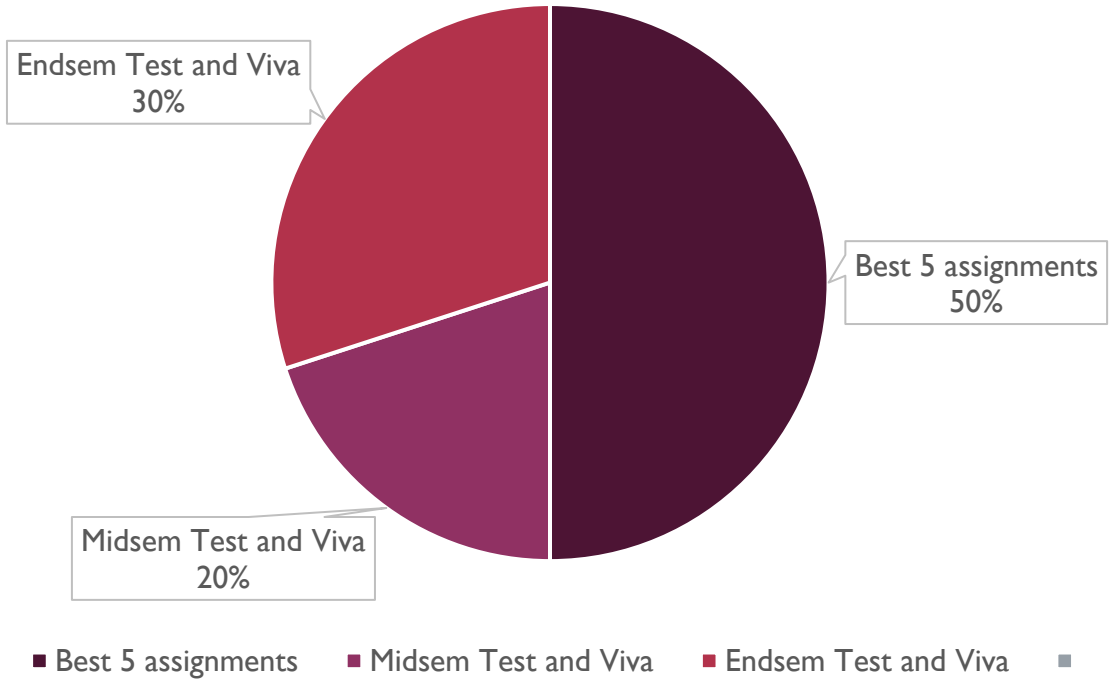
HONOR CODE

- Many algorithms are "*nondeterministically trivial*"
- Please do as much of the work as possible on your own without consulting anyone else or any other outside resources!

- Goal: Learn how to devise your own algorithms
- Always *Cite* the sources (if you have taken ideas from outside sources) Avoid plagiarism

- We will follow ZERO TOLERANCE POLICY if we can find such cases (copying entire solutions from your peers, outside sources, using ChatGPT etc.)

LAB ASSIGNMENTS



ASSIGNMENTS AND QUESTION TYPES

- Grounded in real-world scenarios that require practical solutions *(Questions will simulate actual problems that software engineers and computer scientists face)*
- **No Rote Memorization:** *Understanding concepts will be assessed through their application in solving problems*
- Innovative thinking will be rewarded, especially when it leads to more efficient or elegant solutions ☺
- You will need to consider time and space complexity when designing algorithms.
- **Reflection on Problem-Solving Processes**: You will be asked to **document** and reflect on your problem-solving process (specifically for Lab Assignments and Tutorial lectures)

SYLLABUS

- M1: **How do we reason about algorithms?**
  - Understand the implications of algorithmic choices in real-world applications.
  - **Mathematical Foundations (**basics of complexity analysis)
  - Study common patterns in algorithm behavior and learn to predict performance
  - Master's Theorem
  - Amortized Analysis (Average Running time of algorithms over a sequence of operations.)
  - Rationalizing Algorithms - reasoning about algorithms beyond just their correctness and efficiency **(Design, Behavioural Understanding, Impact, Algorithm Adaptability)**
  - Detail discussion on Insertion Sort

SYLLABUS

- M2: **Fundamental Graph Algorithms.**
  - Traversal Techniques - Depth-First Search (DFS) and Breadth-First Search (BFS). Learn how to apply DFS and BFS to solve problems like cycle detection and level-order traversal.
  - Ordering and Connectivity
    - Topological Sort to understand dependencies and ordering.
    - Analyze Strongly Connected Components to explore deep graph connectivity.
  - Pathfinding Algorithms
    - Dijkstra's Algorithm for single-source shortest paths.
    - Extend pathfinding understanding to all-pair shortest path problems.
  - Optimizing Network Flow

SYLLABUS

- M3: Divide-and-Conquer Algorithms
  - Understanding Recursion in Algorithms
    - Analyze MergeSort to see recursion in action in sorting algorithms.
    - Study recurrence relations as a method to express the runtime of recursive algorithms.
    - Learn about Binary Search as an example of the divide-and-conquer strategy.
  - Priority Queues and Heaps
    - Understand binary heaps as a fundamental data structure for implementing priority queues.
    - Explore Heapsort as an example of a sorting algorithm that leverages a binary heap.
    - Discuss the lower bound of sorting and what it means for the efficiency of sorting algorithms.
  - Divide-and-Conquer Recurrences
  - How do we solve non-uniform recurrences? The Median-of-Medians Algorithm & The Substitution Method

SYLLABUS

- M4: Randomized Algorithm
  - The Power of Randomness!
  - The Substitution Method, Quickselect, Linearity of Expectation
  - Analyzing Randomized Algorithms (QuickSort and applications to find large cuts in a graph, a problem in network design)
  - Confidence in Randomization (Monte Carlo algorithms, Karger's Min-Cut Algorithm)
  - Data Structures and Randomization (Hash Functions Hash Tables)

SYLLABUS

- M5: Greedy Algorithms
  - **Can locally optimal decisions be globally optimal?**
    - Interval Scheduling
    - “Greedy Stays Ahead”
  - **How do you link up nodes at a low cost?**
    - Minimum Spanning Trees
    - Prim's Algorithm
  - **Can we locally modify solutions to build better ones?**
    - Exchange Arguments
    - Dijkstra's Algorithm Revisited
    - Kruskal's Algorithm

SYLLABUS

- M6: Dynamic Programming and Network Path Optimization
  - **Leveraging Previous Calculations**:
    - What is the Linear Independent Set problem, and how can its solutions be reused?
    - What defines Dynamic Programming?
    - How is Sequence Alignment approached using Dynamic Programming?
  - **Routing in Networks:**
    - How do routers on the Internet determine the best paths?
    - How does the Bellman-Ford Algorithm facilitate this?
  - **Shortest Path Precomputation**:
    - Is it possible to precompute shortest paths for all network nodes?
    - How do the Floyd-Warshall and Johnson's Algorithms enable this?

SYLLABUS

- M7: Tackling Intractable Problems and NP-Completeness
- **Approaching Intractable Problems:**
  - How do you handle problems that are computationally intractable? Understand NP-Hardness and its implications.
  - Study the Knapsack Problem and Pseudo polynomial-Time Algorithms.
- **Simplifying Complex Problems:**
  - Can specific instances of hard problems be solved more easily? Explore solving the Independent Set problem on Trees.
- **Estimating Answers to Hard Problems:**
  - Is it possible to guess solutions for complex problems? Learn about Randomized Approximation Algorithms and their application to MAX-3SAT.
- **Understanding NP-Completeness:**
  - What does polynomial-time verification mean in the context of NP-completeness? Reducibility and its role
  - Examine NP-complete problems and the fundamentals of Approximation Algorithms.
- **Exploring the Horizon of Algorithms:** What are the future prospects and areas of study in algorithms?

TURING AWARD WINNERS!

- Edsger Wybe Dijkstra (1972): *For fundamental contributions to programming as a high, intellectual challenge; for eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; for illuminating perception of problems at the foundations of program design.*

TURING AWARD WINNERS!

- **Stephen Arthur Cook (1982):** Recognized for his work on the complexity of computation and, in particular, for his articulation of what is known as the "**P vs NP**" problem, a central question in the theory of computing.

TURING AWARD WINNERS!

- **Richard Manning Karp (1985)**: Awarded for his research in the theory of algorithms, including the development of efficient algorithms for network flow and other combinatorial optimization problems

- Karp's 1972 paper "*Reducibility Among Combinatorial Problems*" proved that many commonly studied combinatorial problems are variants of the same problem, which implies they are all probably intractable (NP-complete problems—that is, problems for which no efficient solution algorithm is known).

TURING AWARD WINNERS!

- **John E. Hopcroft and Robert Endre Tarjan (1986):** Recognized for fundamental achievements in the design and analysis of algorithms and data structures

WHY STUDY ALGORITHMS

- **Set of well-defined rules to solve a computational problem** *(Sorting, Shortest Path, Task scheduling ….)*
- Important for all other branches of computer science (database indices – balanced search tree, computational biology – DP, routing in communication network – shortest path algo …)
- Plays a key role in modern technological innovation
- Provides novel “lens” on processes outside of computer science and technology – quantum mechanics, economic markets, …
- Challenging and Fun!

NEXT LAB CLASS (9TH JAN, TUESDAY)

- Assignments will be on **Heap, Tree, Sorting. Revise these topics and fundamental DS (like Linked List, Queue)**

- **Submission:** Code and Explanation (pdf file)

INTRODUCTION TO RECURSIVE FORMULATIONS FOR ALGORITHM DESIGN

RECURSION

A recursive function is one that calls itself

```
def i_am_recursive(x) :
    maybe do some work
    if there is more work to do :
        i_am_recursive(next(x))
    return the desired result
```

RECURSIVE DEFINITIONS

- Every recursive function definition includes two parts
  - **Base case(s) (non-recursive):** One or more simple cases that can be done right away
  - **Recursive case(s):** One or more cases that require solving "simpler" version(s) of the original problem.
- What do we mean by "**Simple**r"?

EASY EXAMPLE: FACTORIAL

- $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

  $2! = 2 \times 1$

  $3! = 3 \times 2 \times 1$

  $4! = 4 \times 3 \times 2 \times 1$

- ***alternatively:***

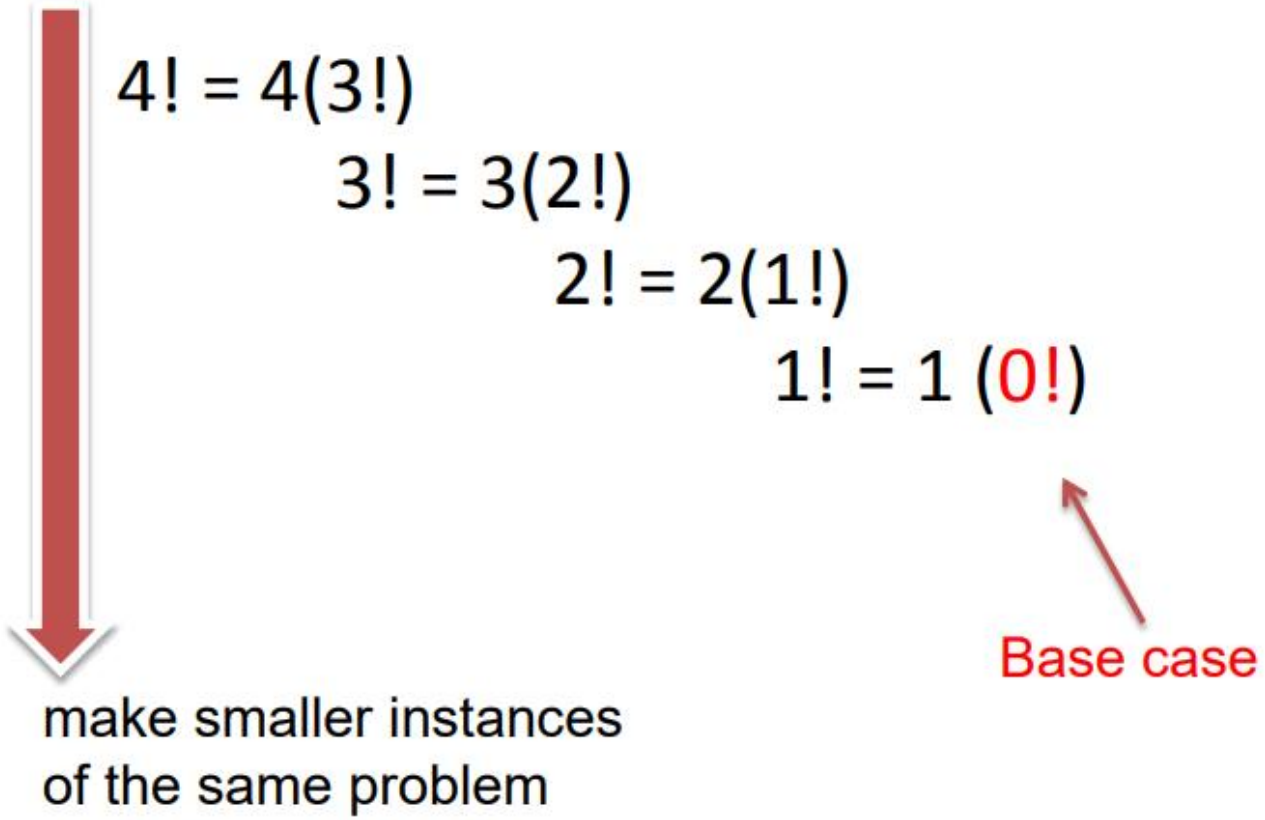
  $0! = 1$ **(Base case)**

  $n! = n \times (n-1)!$ **(Recursive case)**

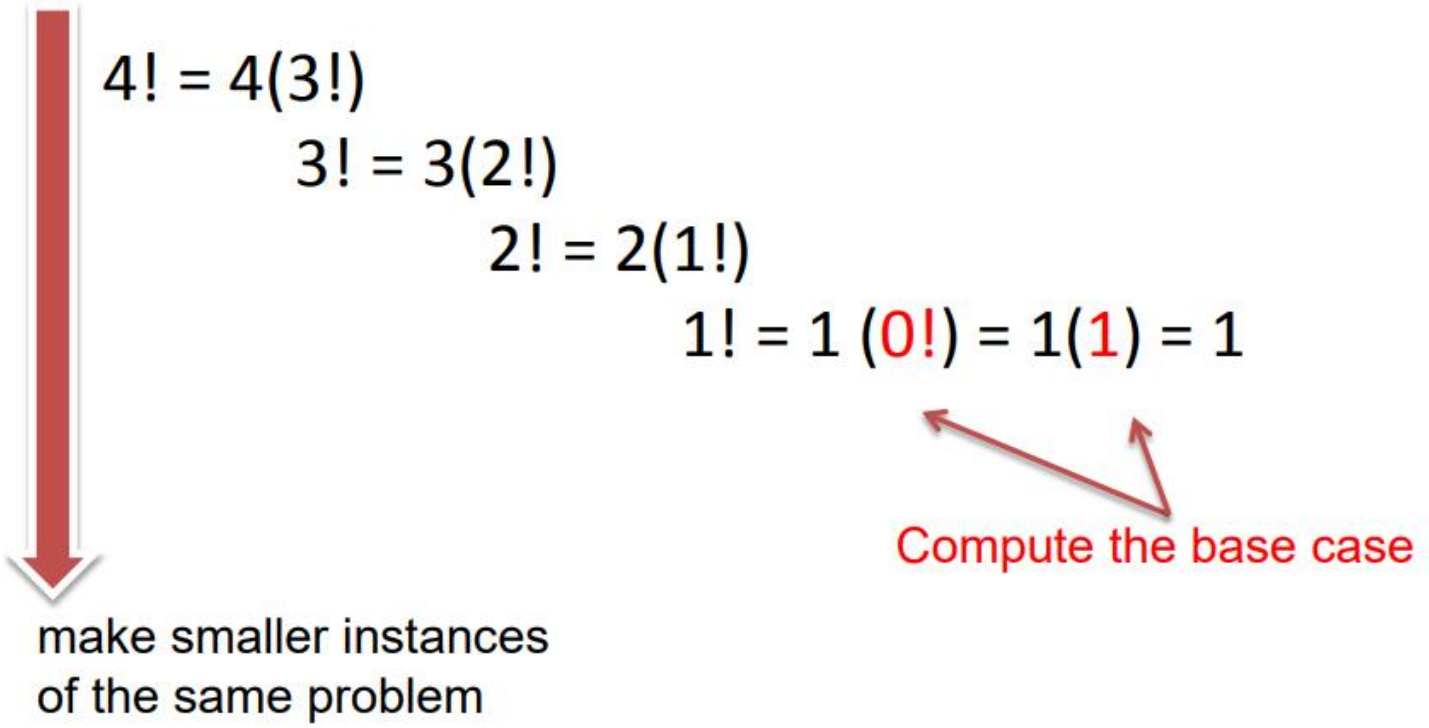
  So $4! = 4 \times 3!$

  $\rightarrow 3! = 3 \times 2! \rightarrow 2! = 2 \times 1! \rightarrow 1! = 1 \times 0!$

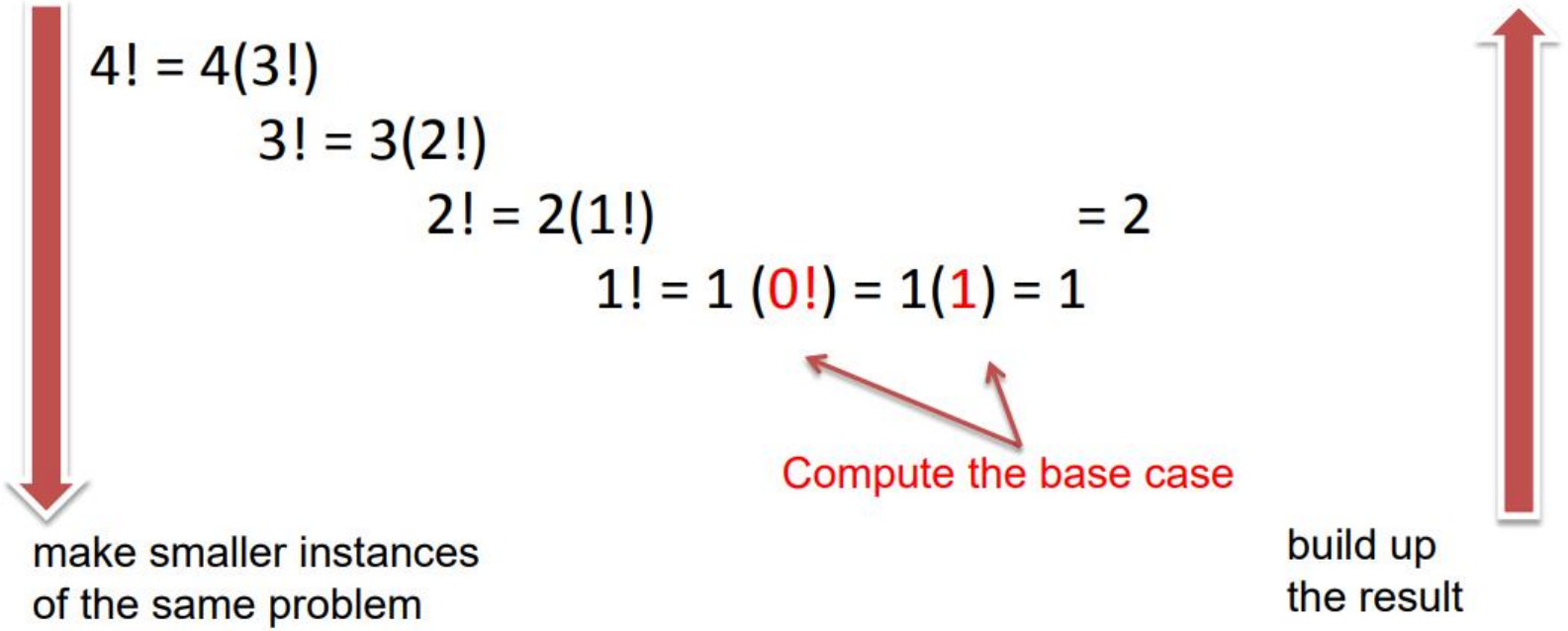
RECURSION CONCEPTUALLY



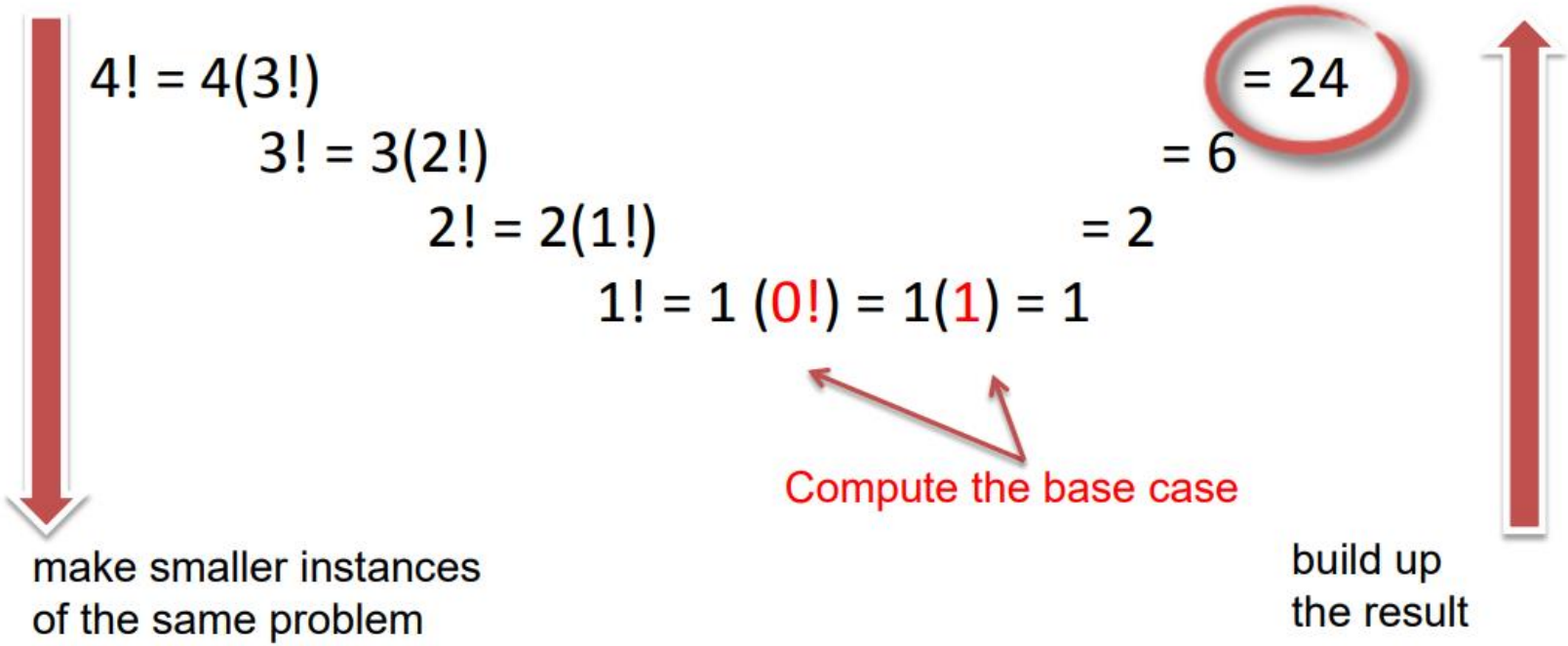
RECURSION CONCEPTUALLY



RECURSION CONCEPTUALLY



RECURSION CONCEPTUALLY



RECURSIVE VS ITERATIVE SOLUTION

- For every recursive function, there is an equivalent iterative solution.
- For every iterative function, there is an equivalent recursive solution.
- But some problems are easier to solve one way than the other way.
- And be aware that most recursive programs need space for the stack, behind the scenes

MULTIPLE RECURSIVE CALLS

- So far we've used just one recursive call to build up our answer
- The real conceptual power of recursion happens when we need more than one!
- Example: **Fibonacci numbers**

RECURSIVE DEFINITION

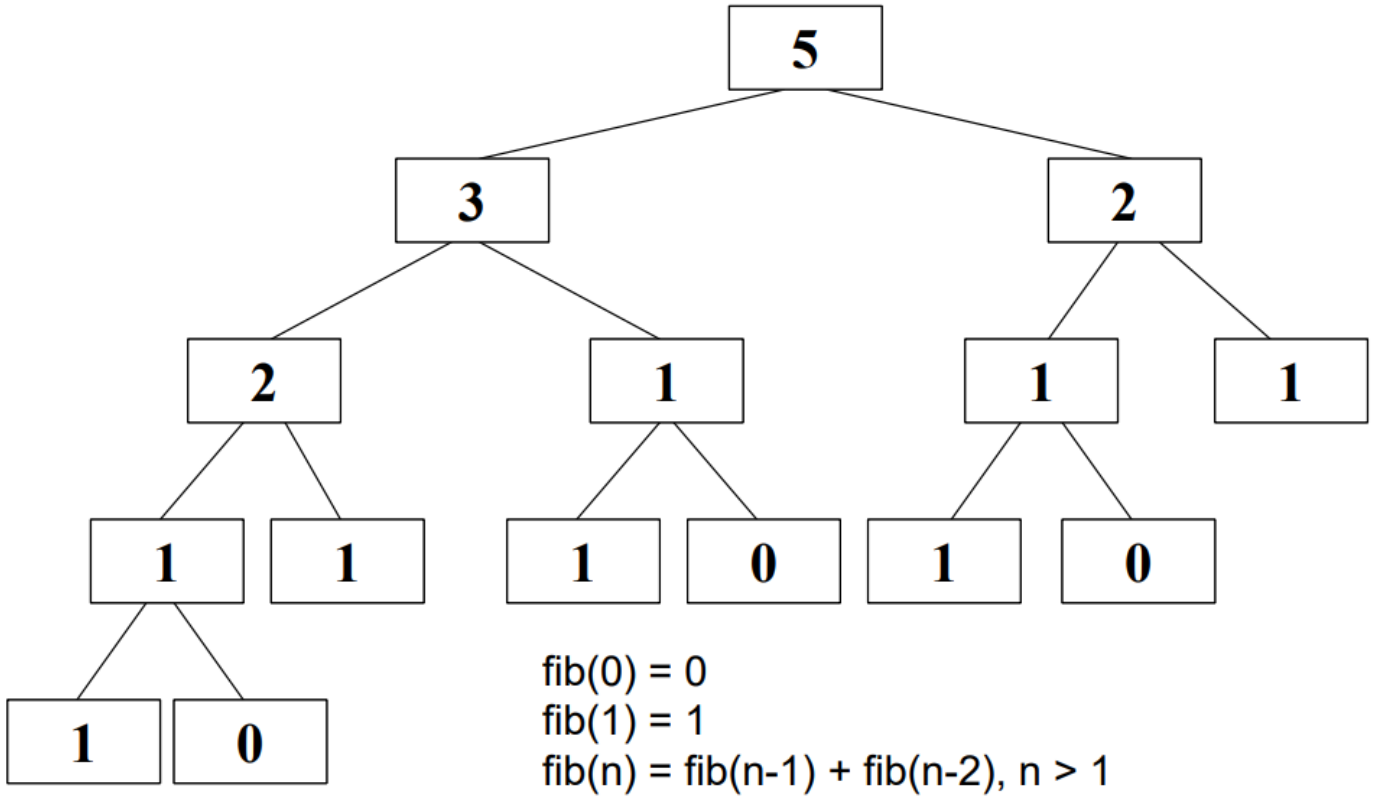
Let fib(n) = the nth Fibonacci number, n ≥ 0

- fib(0) = 0 (base case)
- fib(1) = 1 (base case)
- fib(n) = fib(n-1) + fib(n-2), n > 1

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Two recursive calls!

RECURSIVE CALL TREE



fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1

FIRST PROBLEM: LARGEST OF A SET OF NUMBERS

Sequential comparison (Board work)

LARGEST: ANALYSIS

By induction

- Base condition n=1
- Inductive condition:
  - Correctly for all $n<n_0$
  - We prove inductively it is true for $n_0 + 1$
- Therefore, it is true for all n>=1

ANALYSIS OF ALGORITHMS

- Issues:
  - Correctness
  - Time Efficiency
  - Space Efficiency
  - Optimality
- Approaches:
  - Theoretical Analysis
  - Empirical Analysis

ANALYSIS OF ALGORITHMS

- Correctness – Does it work as advertised?
- Time Efficiency – Are time requirements minimized?
- Space Efficiency – Are space requirements minimized?
- Optimality – Do we have the best balance between minimizing time and space?

THEORETICAL ANALYSIS OF TIME EFFICIENCY

- Time efficiency is analyzed by determining the number of repetitions of the **basic operation** as a function of **input size**
- Basic operation: the operation that contributes most towards the running time of the algorithm

$$T(n) \approx c_{op} C(n)$$

Running Time

Execution Time For Basic Operation

Number Of Times Basic Operation Is Executed

INPUT SIZE AND BASIC OPERATION EXAMPLES

Problem	**Input size measure**	**Basic operation**
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

EMPIRICAL ANALYSIS OF TIME EFFICIENCY

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)
- Count actual number of basic operation's executions
- Analyze the empirical data

BEST-CASE, AVERAGE-CASE, WORST-CASE

- For some algorithms efficiency depends on form of input:
  - Worst case: $C_{worst}(n)$ – maximum over inputs of size n
  - Best case: $C_{best}(n)$ – minimum over inputs of size n
  - Average case: $C_{avg}(n)$ – "average" over inputs of size n

AVERAGE-CASE

- Average case: $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

DIY

Example: Sequential Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- Best case?
- Worst case?
- Average case?

TYPES OF FORMULAS FOR BASIC OPERATION'S COUNT

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\ n^2$

- Formula indicating order of growth with unknown multiplicative constant

  e.g., $C(n) \approx cn^2$

ORDER OF GROWTH

- Most important: **Order of growth within a constant multiple as $n \to \infty$**
- Example: – How much faster will algorithm run on computer that is twice as fast?
- How much longer does it take to solve problem of double input size?

VALUES OF SOME IMPORTANT FUNCTIONS AS N→∞

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}		

ASYMPTOTIC ORDER OF GROWTH

- A way of comparing functions that ignores constant factors and small input sizes
  - **O(g(n))** - class of functions f(n) that grow no faster than g(n)
  - **Θ(g(n))** - class of functions f(n) that grow at same rate as g(n)
  - **Ω(g(n))** - class of functions f(n) that grow at least as fast as g(n)

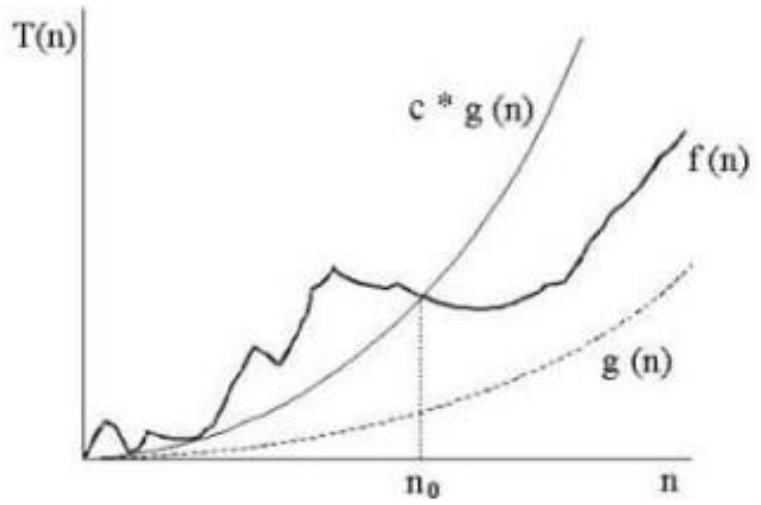
It is called "asymptotic" because it focuses on the behavior of functions as they approach certain limits without necessarily reaching or converging to those limits. This notation is particularly useful for analyzing the performance of algorithms as the input size grows towards infinity

ESTABLISHING ORDER OF GROWTH USING THE DEFINITION

- Definition: $f(n)$ is in $O(g(n))$ if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple), i.e., there exist positive constant c and non-negative integer n_0 such that

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

- Examples:
  - $10n$ is $O(n^2)$
  - $5n+20$ is $O(n)$



BASIC ASYMPTOTIC EFFICIENCY CLASSES

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n or linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Ω-NOTATION (LOWER BOUNDS)

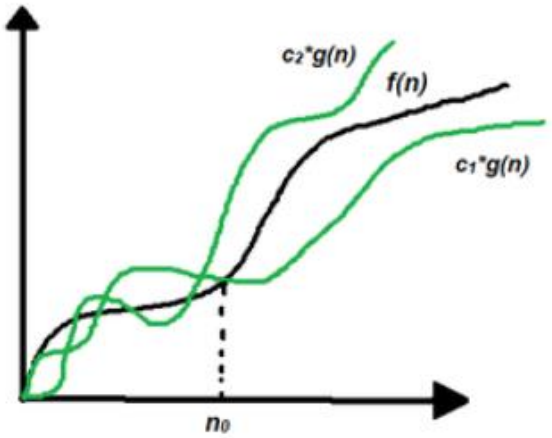
$$\Omega(g(n)) = \{ f(n) : \text{there exist constants } c > 0, n_0 > 0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \}$$

EXAMPLE: $\sqrt{n} = \Omega(\lg n) \quad (c = 1, n_0 = 16)$

Θ-NOTATION (TIGHT BOUNDS)

Θ (g(n)) = {f(n): there exist positive constants c_1, c_2 and n_0 such that

$0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$ *for all $n \ge n_0$}*



EXAMPLE: $\frac{1}{2}n^2 - 2n = \Theta(n^2)$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

SMALL O-NOTATION AND SMALL OMEGA-NOTATION

O-notation and Ω-notation are like $\leq$ and $\geq$.
o-notation and ω-notation are like $<$ and $>$.

$$o(g(n)) = \{ f(n) : \text{for any constant } c > 0, \text{ there is a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \}$$

EXAMPLE: $2n^2 = o(n^3)$ $(n_0 = 2/c)$

SMALL O-NOTATION AND SMALL OMEGA-NOTATION

$$\omega(g(n)) = \{ f(n) : \text{for any constant } c > 0, \text{ there is a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \}$$

EXAMPLE: $\sqrt{n} = \omega(\lg n)$ $\quad (n_0 = 1+1/c)$

HERE'S A TABLE THAT SUMMARIZES THE KEY RESTRICTIONS IN THESE FOUR DEFINITIONS:

Definition	$\boxed{?}\ c > 0$	$\boxed{?}\ n_0 \geq 1$	$f(n)\ \boxed{?}\ c \cdot g(n)$
$O()$	$\exists$	$\exists$	$\leq$
$o()$	$\forall$	$\exists$	$<$
$\Omega()$	$\exists$	$\exists$	$\geq$
$\omega()$	$\forall$	$\exists$	$>$

Let $f(n) = 7n + 8$ and $g(n) = n$. Is $f(n) \in o(g(n))$?

In order for that to be true, for any c, we have to be able to find an n_0 that makes $f(n) < c \cdot g(n)$ asymptotically true.

However, this doesn't seem likely to be true. Both $7n + 8$ and n are linear, and $o()$ defines loose upper-bounds. To show that it's not true, all we need is a counter–example.

Because any $c > 0$ must work for the claim to be true, let's try to find a c that won't work. Let $c = 100$. Can we find a positive n_0 such that $7n + 8 < 100n$? Sure; let $n_0 = 10$. Try again!

Let's try $c = \frac{1}{100}$. Can we find a positive n_0 such that $7n + 8 < \frac{n}{100}$? No; only negative values will work. Therefore, $7n + 8 \notin o(n)$, meaning $g(n) = n$ is not a loose upper-bound on $7n + 8$.

PROPERTIES OF ASYMPTOTIC NOTATIONS

Reflexivity: If f(n) is given then

$$f(n) = O(f(n))$$

Example: If $f(n) = n^3 \Rightarrow O(n^3)$ Similarly,

$$f(n) = \Omega(f(n))$$
$$f(n) = \Theta(f(n))$$

Satisfied by Big-Theta, Big-Oh, Big-Omega

PROPERTIES OF ASYMPTOTIC NOTATIONS

Symmetry:

```
f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))
```

Example: If $f(n) = n^2$ and $g(n) = n^2$ then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

Proof:

- ***Necessary part:*** $f(n) = \Theta(g(n)) \Rightarrow g(n) = \Theta(f(n))$ By the definition of Θ, there exists positive constants c1, c2, no such that $c1.g(n) \leq f(n) \leq c2.g(n)$ for all $n \geq no \Rightarrow g(n) \leq (1/c1).f(n)$ and $g(n) \geq (1/c2).f(n) \Rightarrow (1/c2).f(n) \leq g(n) \leq (1/c1).f(n)$ Since c1 and c2 are positive constants, 1/c1 and 1/c2 are well defined. Therefore, by the definition of Θ, $g(n) = \Theta(f(n))$
- ***Sufficiency part:*** $g(n) = \Theta(f(n)) \Rightarrow f(n) = \Theta(g(n))$ By the definition of Θ, there exists positive constants c1, c2, no such that $c1.f(n) \leq g(n) \leq c2.f(n)$ for all $n \geq no \Rightarrow f(n) \leq (1/c1).g(n)$ and $f(n) \geq (1/c2).g(n) \Rightarrow (1/c2).g(n) \leq f(n) \leq (1/c1).g(n)$ By the definition of Theta(Θ), $f(n) = \Theta(g(n))$

Satisfied by Big-Theta

PROPERTIES OF ASYMPTOTIC NOTATIONS

Transistivity:

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

Example: If f(n) = n, g(n) = n^2 and h(n) = n^3 ⇒ n is O(n^2) and n^2 is O(n^3) then n is O(n^3)

Proof: f(n) = O(g(n)) and g(n) = O(h(n)) ⇒ f(n) = O(h(n)) By the definition of Big-Oh(O), there exists positive constants c, no such that f(n) ≤ c.g(n) for all n ≥ no ⇒ f(n) ≤ c1.g(n) ⇒ g(n) ≤ c2.h(n) ⇒ f(n) ≤ c1.c2h(n) ⇒ f(n) ≤ c.h(n), where, c = c1.c2

By the definition, f(n) = O(h(n)) Similarly,

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

Satisfied by Big-Theta, Big-Omega, Big-Oh, little o, little omega

PROPERTIES OF ASYMPTOTIC NOTATIONS

Transpose Symmetry:

```
f(n) = O(g(n)) if and only if g(n) = Ω(f(n))
```

Example: If $f(n) = n$ and $g(n) = n^2$ then n is $O(n^2)$ and n^2 is $\Omega(n)$

Proof:

- ***Necessary part:*** $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$ By the definition of Big-Oh (O) $\Rightarrow f(n) \leq c.g(n)$ for some positive constant c $\Rightarrow g(n) \geq (1/c).f(n)$ By the definition of Omega (Ω), $g(n) = \Omega(f(n))$
- ***Sufficiency part:*** $g(n) = \Omega(f(n)) \Rightarrow f(n) = O(g(n))$ By the definition of Omega (Ω), for some positive constant c $\Rightarrow g(n) \geq c.f(n) \Rightarrow f(n) \leq (1/c).g(n)$ By the definition of Big-Oh(O), $f(n) = O(g(n))$

WORST CASE, AND BEST CASE ANALYSIS

- Linear Search
- Binary Search Tree

INSERTION SORT

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

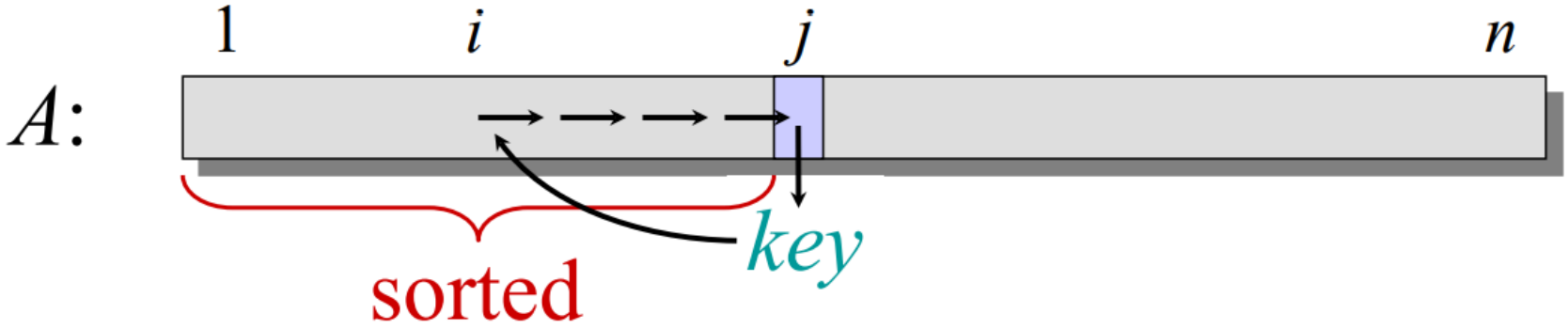
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

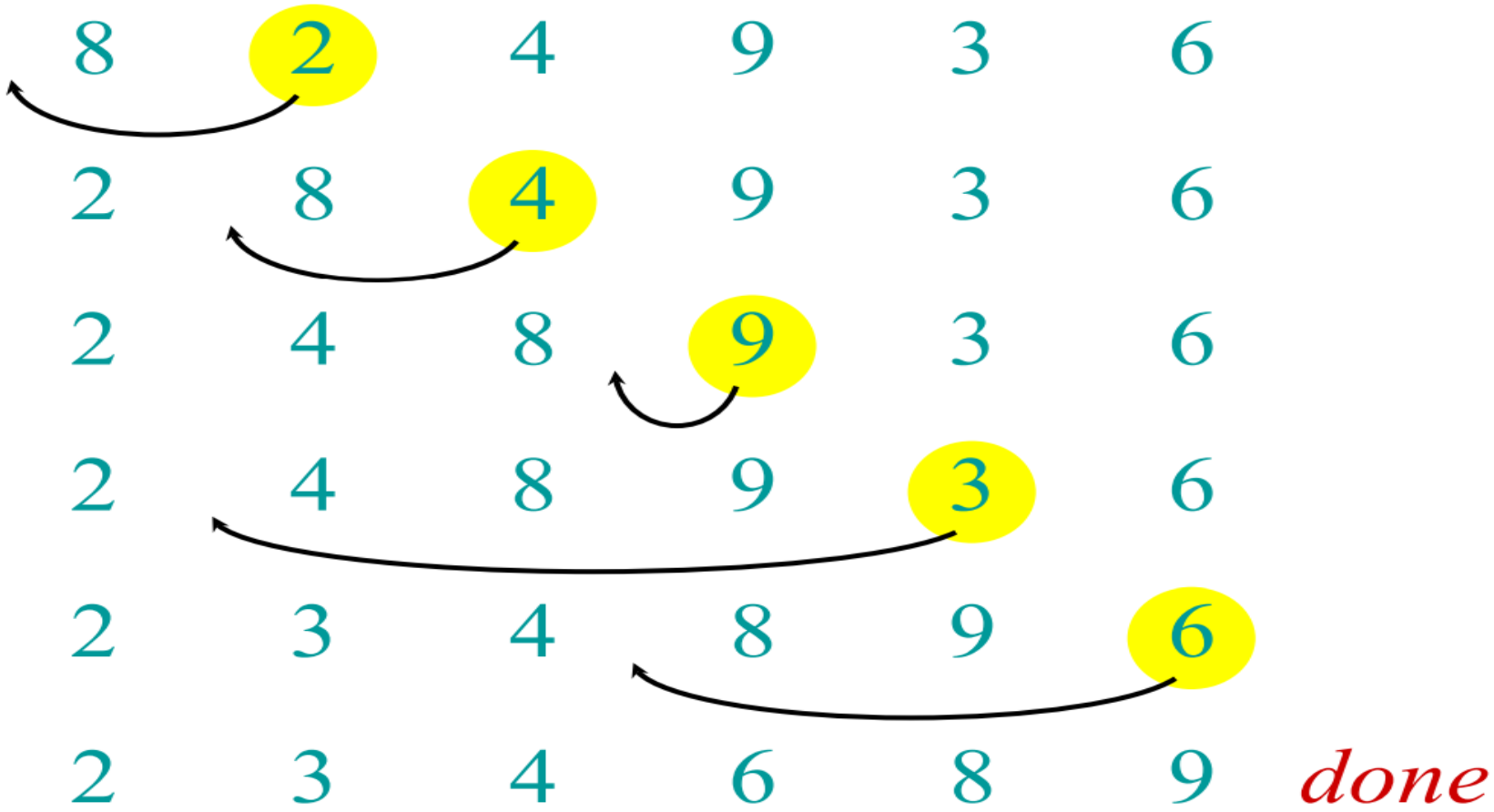
INSERTION SORT

"pseudocode"

```
INSERTION-SORT (A, n)      ▷ A[1 . . n]
    for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
           A[i+1] = key
```



EXAMPLE



PROOF OF CORRECTNESS

To prove insertion sort is correct, we will use "loop invariants." The loop invariant we'll use is

Lemma: At the start of each iteration of the for loop, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

To use this, we need to prove three conditions:

Initialization: The loop invariant is satisfied at the beginning of the for loop.

Maintenance: If the loop invariant is true before the ith iteration, then the loop invariant will be true before the $i+1$st iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Note that this is basically mathematical induction (the initialization is the base case, and the maintenance is the inductive step).

INSERTION SORT

Initialization: Before the first iteration (which is when $j = 2$), the subarray $[1..j-1]$ is just the first element of the array, $A[1]$. This subarray is sorted, and consists of the elements that were originally in $A[1..1]$.

Maintenance: Suppose $A[1..j-1]$ is sorted. Informally, the body of the for loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$ and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4-7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.

Termination: The condition causing the for loop to terminate is that $j > n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. By the initialization and maintenance steps, we have shown that the subarray $A[1..n+1-1] = A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order.

DIY Problem:

Best, worst, and average case analysis of Insertion Sort

NOTE: Average-case analysis presupposes a probability distribution on inputs, we assume all permutations of a given input array are equally likely

DIVIDE AND CONQUER DESIGN PARADIGM

1. ***Divide*** the problem (instance) into subproblems.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

BINARY SEARCH

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

BINARY SEARCH

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

BINARY SEARCH

Example: Find 9

RECURRENCE FOR BINARY SEARCH