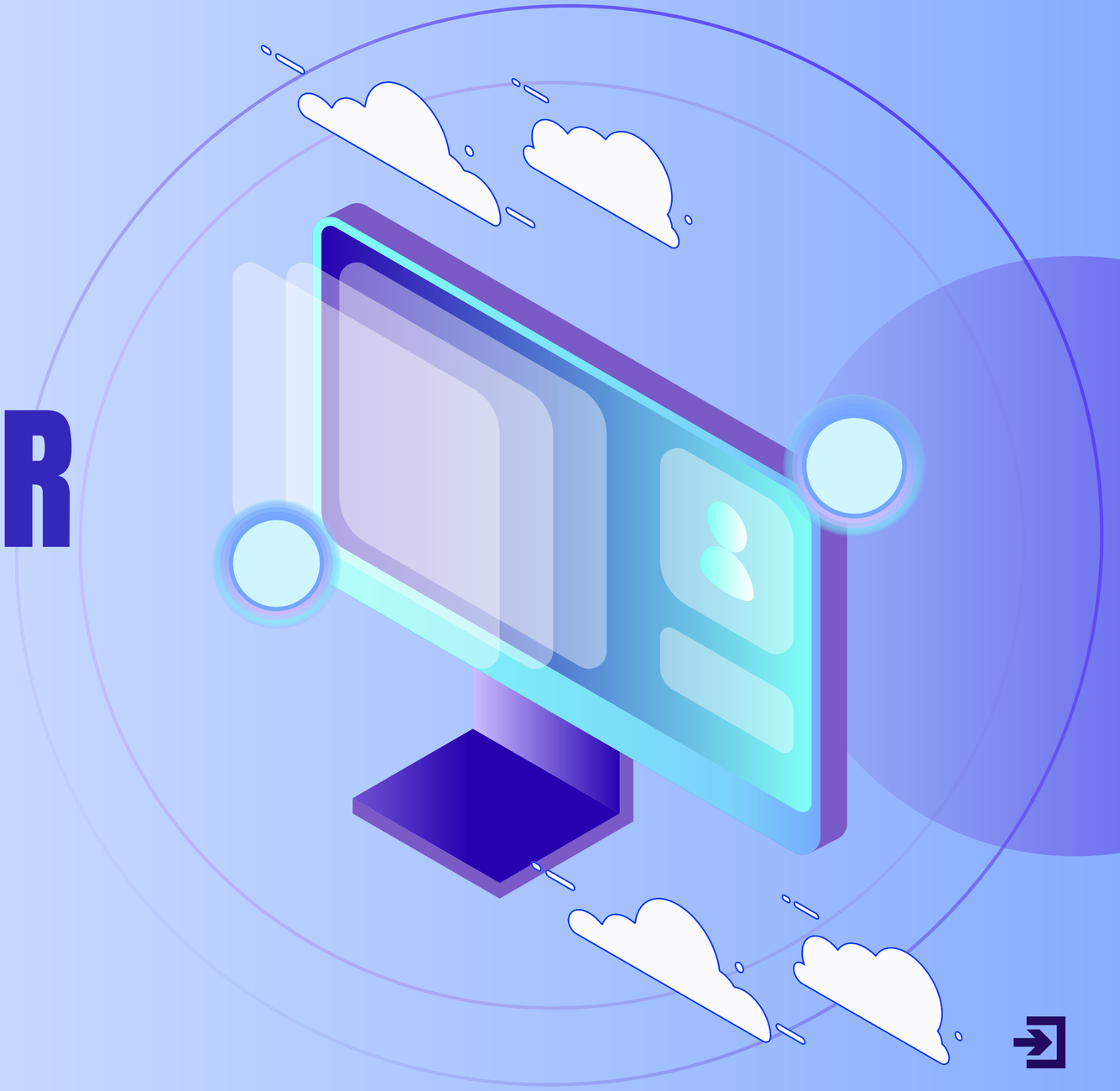
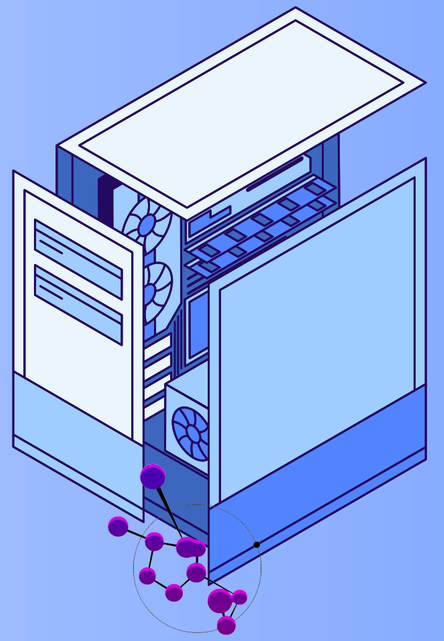
DIVIDE & CONQUER

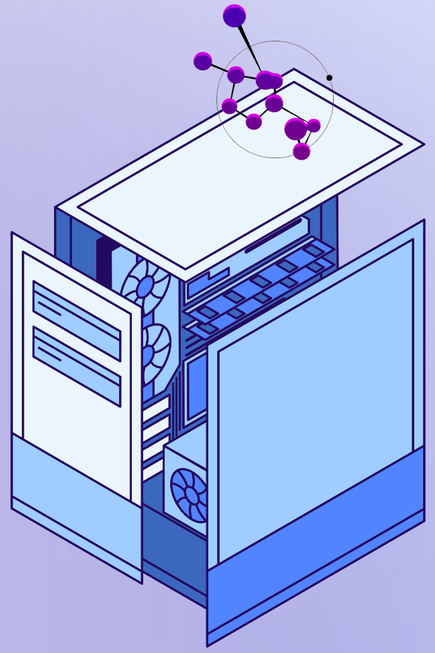
PROOFS



GOALS:



- Understand what to prove (correctness + runtime)
- Practice using induction and recurrence relations
- Apply these ideas to Binary Search



WHAT IS DIVIDE & CONQUER?

- Divide the problem into smaller subproblems
- Conquer each subproblem recursively
- Combine the results

Examples:

- Binary Search
- Merge Sort
- Quick Sort
- Closest Pair of Points

TWO KINDS OF PROOFS

Type	What You're Showing	Example Statement
Correctness Proof	The algorithm gives the right answer for all valid inputs.	"Merge Sort always returns a sorted list containing the same elements."
Runtime Proof	The algorithm's running time matches a certain function (like O(n log n)).	"Merge Sort runs in O(n log n)."

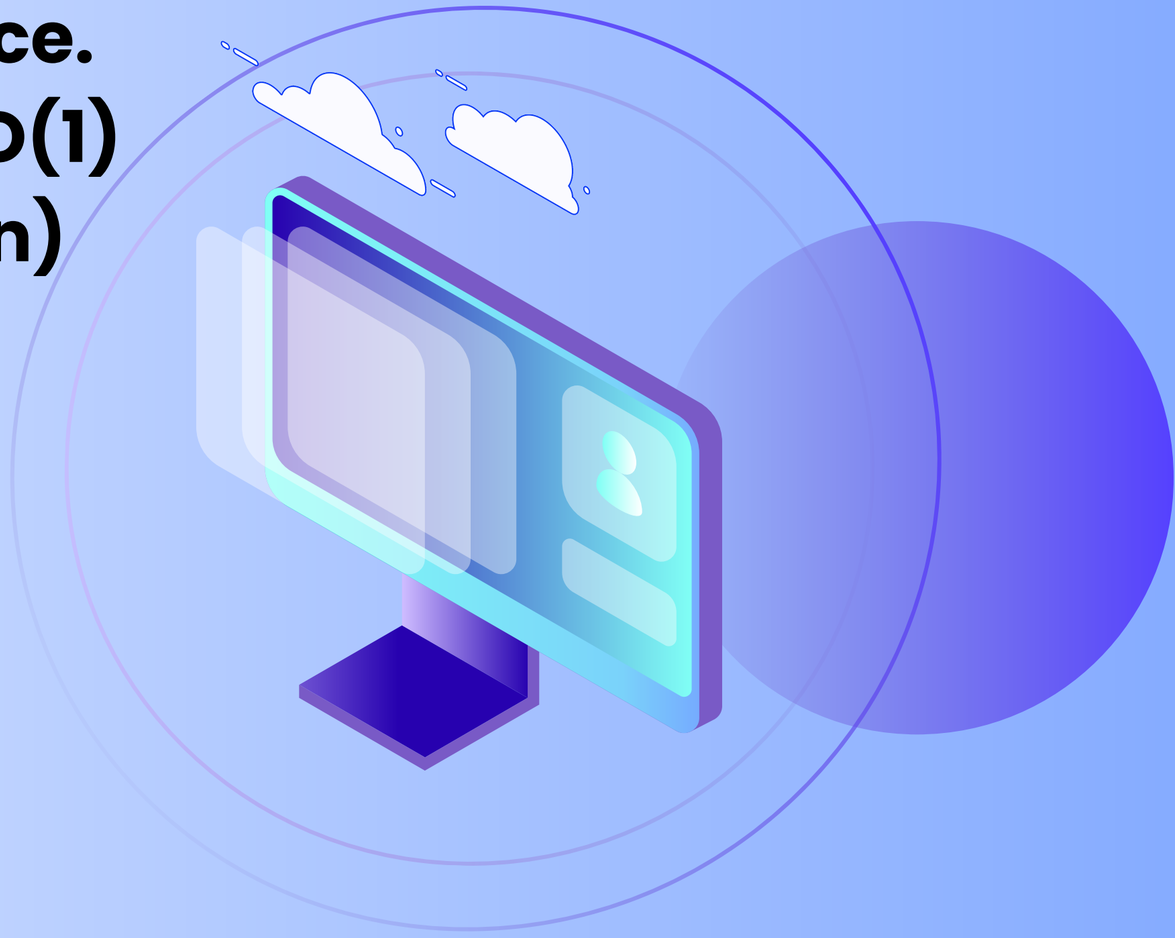
PROVING CORRECTNESS: INDUCTION ON INPUT SIZE

We use mathematical induction because divide-and-conquer algorithms are recursive.

1. **Base Case:** Show that the algorithm works for the smallest possible input (e.g., a list of length 1).
2. **Inductive Hypothesis:** Assume the algorithm works for smaller inputs (size < n).
3. **Inductive Step:** Show it must also work for input of size n using the recursive structure.

PROVING RUNTIME: RECURRENCE RELATIONS

- Express the runtime as a recurrence.
  - Binary Search: $T(n) = T(n/2) + O(1)$
  - Merge Sort: $T(n) = 2T(n/2) + O(n)$
- Solve the recurrence using:
  - Recursion Tree
  - Master Theorem
  - Repeated Substitution
- Conclude asymptotic complexity.
  - Binary Search → $O(\log n)$
  - Merge Sort → $O(n \log n)$



Deep Dive on Proofs & Why It Matters

Day 2 Overview

PART 1

PROOFS IN THE PRACTICAL SENSE

- WHEN ALGORITHMS FAIL PEOPLE: HEALTHCARE.GOV LAUNCH
- TODAYS MISSION: PROVING OUR BINARY SEARCH

PART 2

CORRECTNESS PROOF

- THE “WHY” BEHIND THE CODE
- PROVING CORRECTING STEP BY STEP
- GROUP ACTIVITY: PROVE THE SCENARIO

PART 3

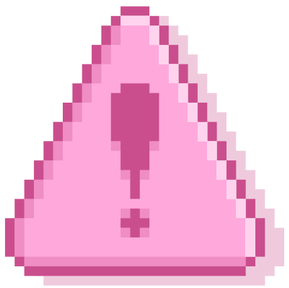
RUNTIME PROOF

- RECURRENCE RELATION
- CIVIC IMPACT OF EFFICIENCY
- GROUP ACTIVITY: PROVE THE SCENERIO

Welcome to

HEALTHCARE.GOV

WHY PROOFS MATTER



Start

Background

Home Content Contact

On October 1st, 2013, the Healthcare.gov launch failed due to inefficient algorithms, lack of testing, insufficient server capacity, and poor project management. Millions were hoping to enroll, but only 6 people were successful in receiving the correct plan. Users experienced problems with "wrong" insurance enrollments, like mistakes in application data and files being sent to the incorrect insurer. These issues stemmed from errors where enrollment information was duplicated or sent to the wrong insurance company.

Event → Healthcare.gov launch 2013

Algorithms → correct bugs

Result → Thousands to millions of people got wrong insurance plans or no insurance

Impact → People couldn't access affordable healthcare they paid for

The Goal: Help millions of Americans find and enroll in health insurance.

The Reality: On day one, only 6 people succeeded. What Went Wrong?

Enrollment data was sent to the wrong insurance companies. People were matched with incorrect or no insurance plans. The system couldn't handle the scale.

- A bug in sorting = wrong order in resource allocation
- A flaw in search = people can't find emergency services
- In civic tech, algorithm errors hurt real people

CORRECTNESS & RUNTIME

1. Formal Proof of Correctness
   Step-by-Step proof for Binary Search and Merge Sort

2. Solving Recurrence Relations
   A detailed look into Three Methods (Recursion Tree, Repeated Substitution, and the Master Theorem for both $T(n) = T(n/2) + c$ and $T(n) = 2T(n/2) + cn$

3. Asymptotic Conclusion
   Rigorously concluding $O(\log n)$ and $O(n \log n)$

Binary Search Skeleton Code


```
def binary_search(arr, target):
    # Initialize search boundaries
    low = ____
    high = ____

    # Continue searching while the search space is
valid
    while ____ <= ____:
       # Find the middle index
       mid = ____
       # Check if we found the target
        if arr[mid] == ____:
            return ____
       # If target is in the right half
        elif arr[____] < ____:
            low = ____
       # If target is in the left half
        else:
            high = ____
     # Target not found
     return ____
```

Binary Search Algorithm Solution


```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
       mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```