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

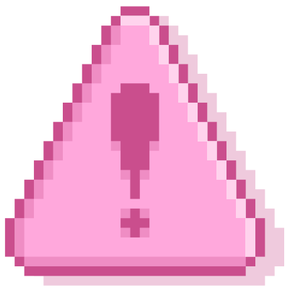
RUNTIME PROOF

- 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
```

Binary Search Proof of Correctness with Induction

Statement F(n): Binary Search correctly returns the index of the array of size n, or -1 if it is not present

Base Case: F(1)

- Array has one element low = 0, high = 0
- The loop runs. mid = 0
- It checks if arr[0] == target
  - If yes, returns 0.
  - If no, it updates low or high and the loop ends, returning -1.

Inductive Hypothesis: Assume F(k) is true for all k < n

- We trust that binary_search works perfectly on any list smaller than the one we have now. This is our foundation.

Inductive Step: Prove F(n) for n > 1

- The algorithm computes mid. Now, three things can happen →

Binary Search Proof of Correctness with Induction

Case 1
arr[mid] == target

- It returns mid immediately. This is trivial but correct.

Case 2
arr[mid] > target

- Because the list is sorted by, say, zip code, every element from mid to the end is too large
- So, it sets high = mid - 1. The new search space is arr[low...mid-1].
- The size of this subarray is mid - low. Since n > 1 and mid is a valid index, this size is less than n
- By our Inductive Hypothesis, the search on this smaller subarray is correct

Case 3
arr[mid] < target

- Symmetric logic. Every element from the start to mid is too small
- It sets low = mid + 1. The new search space is arr[mid+1...high], which has size high - mid, which is also less than n
- The Inductive Hypothesis guarantees correctness here too

Conclusion

- By the principle of mathematical induction, since it's true for n=1 and truth for all k < n implies truth for n, then F(n) is true for all positive integers n.

The Master Theorem gives us a solution. For recurrences of form: T(n) = aT(n/b) + f(n)

- a ≥ 1: Number of recursive calls
- b > 1: Factor by which problem shrinks
- f(n): Work to divide/combine

Taking an equation of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$ and $f(n) > 0$

The Master's Theorem states:

- **CASE 1** - if $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- **CASE 2** - if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
- **CASE 3** - if $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Three Cases:

1. If f(n) grows slower than n^(log_b a) -> Case 1
2. If f(n) grows same as n^(log_b a) -> Case 2
3. If f(n) grows faster than n^(log_b a) -> Case 3

Apply to Binary Search:

- a = 1, b = 2, f(n) = c -
- n^(log_b a) = n^(log_2 1) = n^0 = 1
- f(n) = c = **Θ**(1) = **Θ**(n^0) -> Case 2
- Solution: T(n) ∈ **Θ**(log n)

Apply to Merge Sort:

- a = 2, b = 2, f(n) = cn
- n^(log_b a) = n^(log_2 2) = n^1 = n
- f(n) = cn = **Θ**(n) -> Case 2
- Solution: T(n) ∈ **Θ**(n log n)

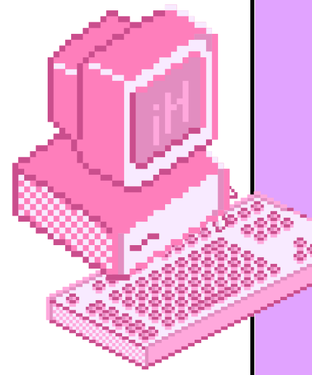
Solving Recurrences - Subsitution - Binary Search

For recurrences where the problem size shrinks quickly, repeated substitution is very effective like in Binary Search.

1. T(n) = T(n/2) + c
2. Substitute: T(n) = [T(n/4) + c] + c = T(n/4) + 2c
3. Substitute: T(n) = [T(n/8) + c] + 2c = T(n/8) + 3c
4. After k substitutions: T(n) = T(n / 2^k) + k*c

Now, we hit the base case. Let T(1) = a (a constant).
Set n/2^k = 1 → n = 2^k → k = $\log_2(n)$

Substitute back:
T(n) = T(1) + log_2(n) * c T(n) = a + c*log_2(n)
Therefore, T(n) ∈ O(log n).



Solving Recurrences - Recursion Tree - Binary Search

Home Content Contact

Recursion Tree Analysis:

- Level 0: 1 problem of size n -> Work = c
  - Searching all shelters in the state
- Level 1: 1 problem of size n/2 -> Work = c
  - Eliminated half the state, searching one region
- Level 2: 1 problem of size n/4 -> Work = c
  - Eliminated half again, searching a county

...

- Level k: 1 problem of size n/2^k -> Work = c
  - Now searching a specific neighborhood

How many levels? → The search stops when the problem size is 1 (when we're looking at a single shelter): n/2^h = 1 → 2^h = n -> h = log_2(n)

Total Work: We have $\log_2(n)$ levels, each doing constant c work. T(n) = c + c + c + ... + c [for log_2(n) terms] T(n) = c * log_2(n)

Therefore, T(n) ∈ O(log n).

THE RECURRENCE RELATIONS

Binary Search

$$T(n) = T(n/2) + c$$

- c represents the constant work: calculating mid, comparing arr[mid] to the target, and updating pointers

Merge Sort

$$T(n) = 2T(n/2) + cn$$

- 2T(n/2) is the cost of solving the two subproblems.
- cn is the cost of merging two sublists of size n/2. This is O(n).