Lecture 12: Problem Paradigms & Complexity Classes

Contents to cover:

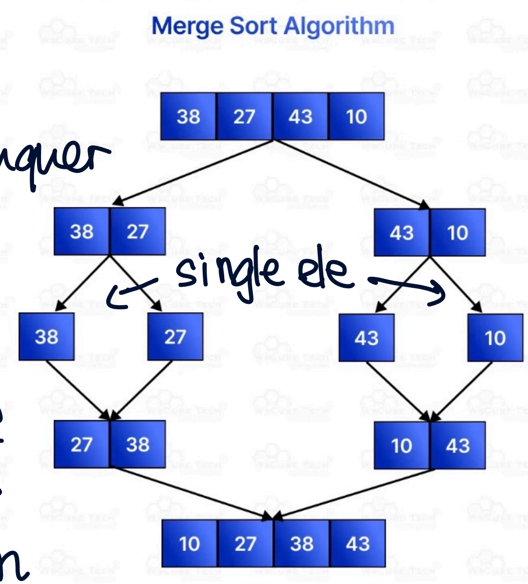
- ~~Divide and conquer~~
- ~~Important: Stimulation & Randomization~~
  ~~Monte Carlo method for calculating π~~
- ~~Approximation~~
- Optimization
- Generate-and-Test (L11)
- Understanding the NP problem } → Not assessable
- Algorithms classifications }

① Divide-and-Conquer

Merge Sort follows the divide-and-conquer paradigm (pattern):

- Divide: split the array into halves at midpoint
- Conquer: Recursively sort each half until you reach single elements
- Merge: Merge the sorted halves in sorted order

single ele

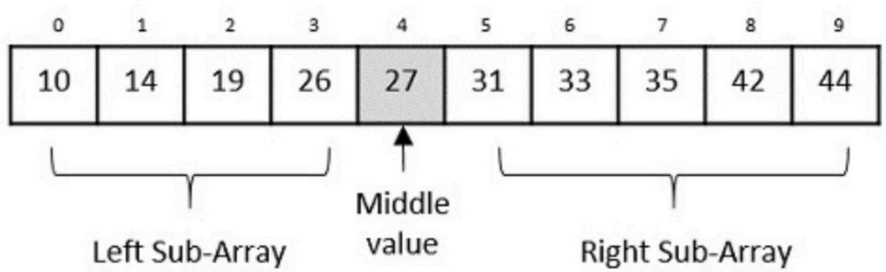


. Has consistent, guaranteed time complexity of $O(n \log n)$

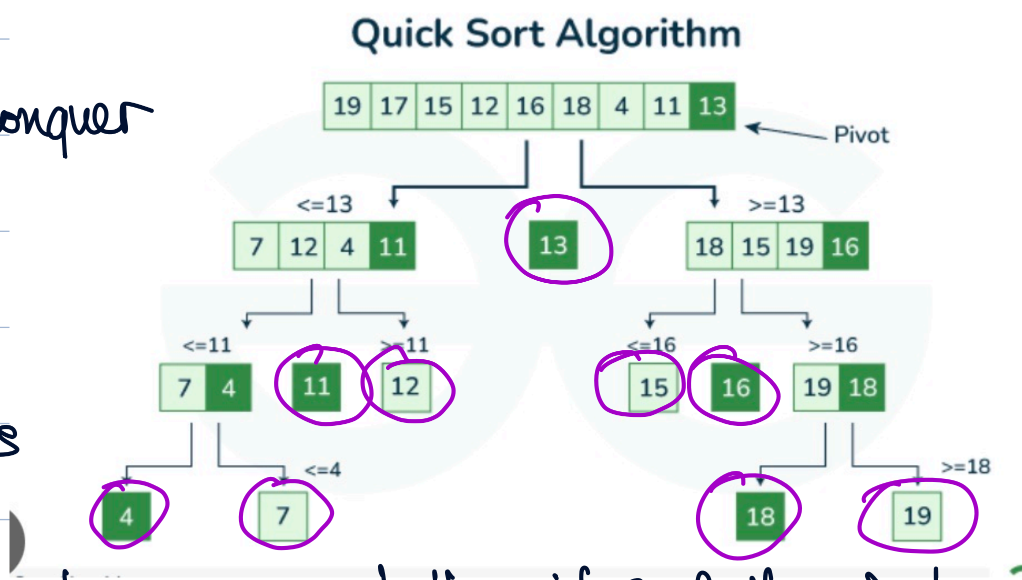
$O(n \log n) = O(n)$ operations * $O(\log n)$ levels

Binary Search ($\neq$ Binary Search Tree – BST) works on sorted arrays

- How Binary search works:
  - Check the middle element of the sorted array (ASC)
  - if
    - target == middle element → return the index
    - target < middle element → return the left half
    - target > middle element → return the right half
  - recurse until found or search space is empty
  - Time complexity:
    - $O(1)$ – Best case
    - $O(\log n)$ – Average & Worst case



Quick sort follows the divide-and-conquer paradigm



- Pivot (Last element)
- Partition: Arrange the array s.t. smaller elements come before it; larger elements come after it
- Recursively sort: Apply quicksort to the subarrays on both sides of the pivot
- Combine

· Time complexity:
- Best & Average case: $O(n \log n) = O(\log n)$ level $* \; O(n)$ elements
- Worst case: $O(n^2)$ – the pivot is always the smallest or largest element in the array

⇒ Unbalanced partition: 1 side as n element, 1 side has 0 element

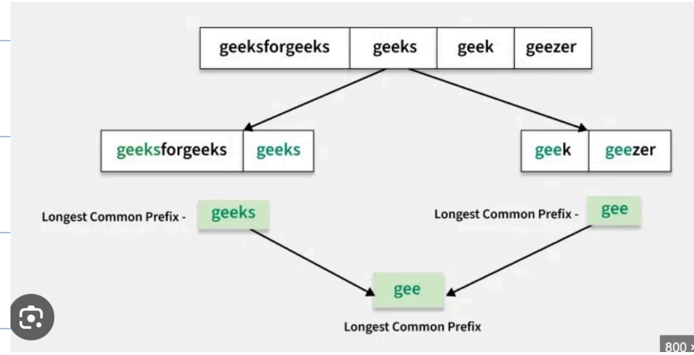
Longest Common prefix (LCP)
- Average: $O(\log n)$ due to recursion stack
- Worst: $O(m * n)$ m–string; n–shortest string

Divide: Split the array of strings into 2 halves

Conquer: Recursively find the longest common prefix for each half

Combine: Compare the 2 prefixes char by char to find the common prefix

- Vital in bioinformatics for DNA and protein sequence analysis



n^{th} Fibonacci # : Time complexity $O(2^n)$ — Not factorial time

```c
#include <stdio.h>

// Function to calculate the nth Fibonacci number using recursion
int nthFibonacci(int n){
    // Base case: if n is 0 or 1, return n
    if (n <= 1){
        return n;
    }
    // Recursive case: sum of the two preceding Fibonacci numbers
    return nthFibonacci(n - 1) + nthFibonacci(n - 2);
}
```

Note: Factorial time complexity will occur if you were generating all possible permutations. For Fibo, it just makes overlapping calls

Fibo problem using Dynamic Programming (DP) - solves complex problem by breaking them into simpler subproblems and storing the result to avoid redundancy

2 main approaches
- Memoization (Top down): uses recursion with a cache
  original problem → Base case
- Tabulation (Bottom up): uses iteration to build solutions from smallest subproblems upwards

Common DP problems
- Fibonacci
- Knapsack ?
- LC subsequences
- Matrix chain

Fibo - Tabulation/ Bottom-Up

```c
#include <stdio.h>

// Function to calculate the nth Fibonacci number
// using iteration
int nthFibonacci(int n) {

    // Handle the edge cases
    if (n <= 1) return n;

    // Create an array to store Fibonacci numbers
    int dp[n + 1];

    // Initialize the first two Fibonacci numbers
    dp[0] = 0;
    dp[1] = 1;

    // Fill the array iteratively
    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    // Return the nth Fibonacci number
    return dp[n];
}
```

② Stimulation & Randomisation

Uses random number generation to solve problems that's diffult/expensive to solve analytically

Monte Carlo π estimation:

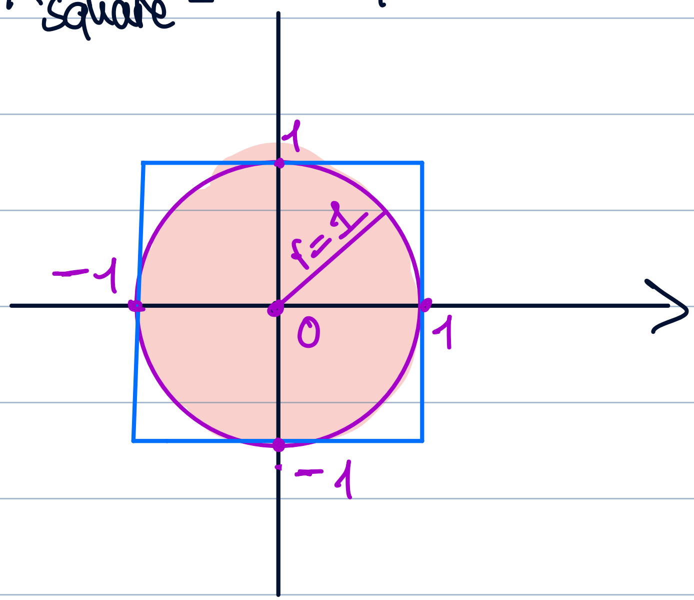
Mathematical foundation:

- Consider a unit circle inscribed in a square
- $\frac{\text{circle area}}{\text{square area}} = \frac{\pi}{4}$
- Random points falling inside circle vs. total point: $\frac{\pi}{4}$

Algorithm (Pseudocode)

$A_{circle} = \pi r^2 = \pi 1^2 = \pi$

$A_{square} = 2^2 = 4$




```c
for (many iterations) {
    x = random number between -1 and 1
    y = random number between -1 and 1

    if (x² + y² <= 1) {
        points_inside_circle++
    }
    total_points++
}

estimated_pi = 4 * (points_inside_circle / total_points)
```

Convergence: As you increase the number of random samples, your estimate approaches the true value of π 2 3 .

$\therefore \frac{\text{circle area}}{\text{square area}} = \frac{\pi}{4}$

$\therefore \pi = 4 * \frac{\text{circle area}}{\text{square area}}$

Monte Carlo vs. Las Vegas Algorithm

Monte Carlo	Las Vegas
· Can run forever · No guarantee of correct answer · Provide approximate / probabilistic results · Used when approximate ans is ok, statistical sampling, simulation problems	· Always terminate w/ finite runtime · Always correct when terminate · Use randomness to improve average-case performance. Eg: + Randomised qsort + Randomised BST (to maintain balance) + ... primality testing (certain variants)

③ Approximation solve a simpler version of the problem that approximates the original with a known and converging error

→ Numerical analysis