Project made by
Nour Charfeddine
Nader Ben Ammar
Mohamed Arbi Ghanmi

DECREASE AND CONQUER ALGORITHM DESIGN STRATEGY

AGENDA

01 Introduction

02 Understanding Decrease and Conquer

03 Recursive Approach

04 Iterative Approach

AGENDA

05 Applications of Decrease and Conquer

06 Advantages and Limitations

07 Best Practices and Tips

08 Conclusion

1/INTRODUCTION

INTRODUCTION

Welcome to our exploration of Decrease and Conquer! This strategy simplifies complex problems by breaking them into smaller parts, conquering each one, and then combining the solutions. Today, we'll dive into its essence, applications, and transformative impact. Join us as we unlock the secrets of Decrease and Conquer and revolutionize problem-solving together!

2/UNDERSTANDING DECREASE AND CONQUER

1/ DECOMPOSITION

Decomposition: The first step involves decomposing the original problem into smaller instances of the same problem. By reducing the size of the problem, we make it more tractable.

2/ SOLVE

Next, we solve each subproblem independently. This may involve applying the same strategy recursively to further decrease the size of the subproblems.

3/ COMBINE

Finally, we combine the solutions of the subproblems to obtain the solution to the original problem.

3/ RECURSIVE APPROACH

RECURSIVE APPROACH

1. Implementing Decrease and Conquer Recursively
2. Step-by-Step Breakdown of Recursive Algorithm
3. Examples Demonstrating Recursive Solutions

Through these sub-topics, we will delve into the application of recursion within the Decrease and Conquer strategy, highlighting its effectiveness in addressing diverse problems with high and efficiency.

BASIC STRUCTURE

DECOMPOSITION

- Start by breaking down the problem into smaller instances, often leveraging recursive function calls.
- **Base Case Identification**: Define the base case, a crucial condition that dictates when the recursion should terminate.
- **Recursive Case Handling**: Address the recursive case, where the problem is reduced to simpler instances until reaching the base case.

WALKTHROUGH OF A RECURSIVE ALGORITHM

- Understand how the problem is divided into smaller parts.
- Track the progression of the algorithm as it recursively solves each subproblem.
- Witness the termination of recursion upon reaching the base case.

EXAMPLES

- **Binary Search**: Experience how recursion simplifies binary search by repeatedly dividing the search space in half until finding the target element.
- **Merge Sort**: Explore the recursive nature of Merge Sort, where the array is recursively divided into smaller subarrays, sorted, and merged back together.

4/ ITERATIVE APPROACH

ITERATIVE APPROACH

The Iterative Approach in Decrease and Conquer algorithm design offers a non-recursive method for solving problems, utilizing loops and iterative constructs. Let's compare it with recursion and showcase two classic examples: Binary Search and Selection Sort.

RECURSIVE APPROACH VS ITERATIVE APPROACH

1. ***Recursive Approach:*** **Solves problems by breaking them down into smaller subproblems, potentially leading to stack overflow for deeply nested calls.**

2. ***Iterative Approach:*** **Uses loops without additional function calls or stack memory, sometimes offering more efficiency and less stack overflow risk.**

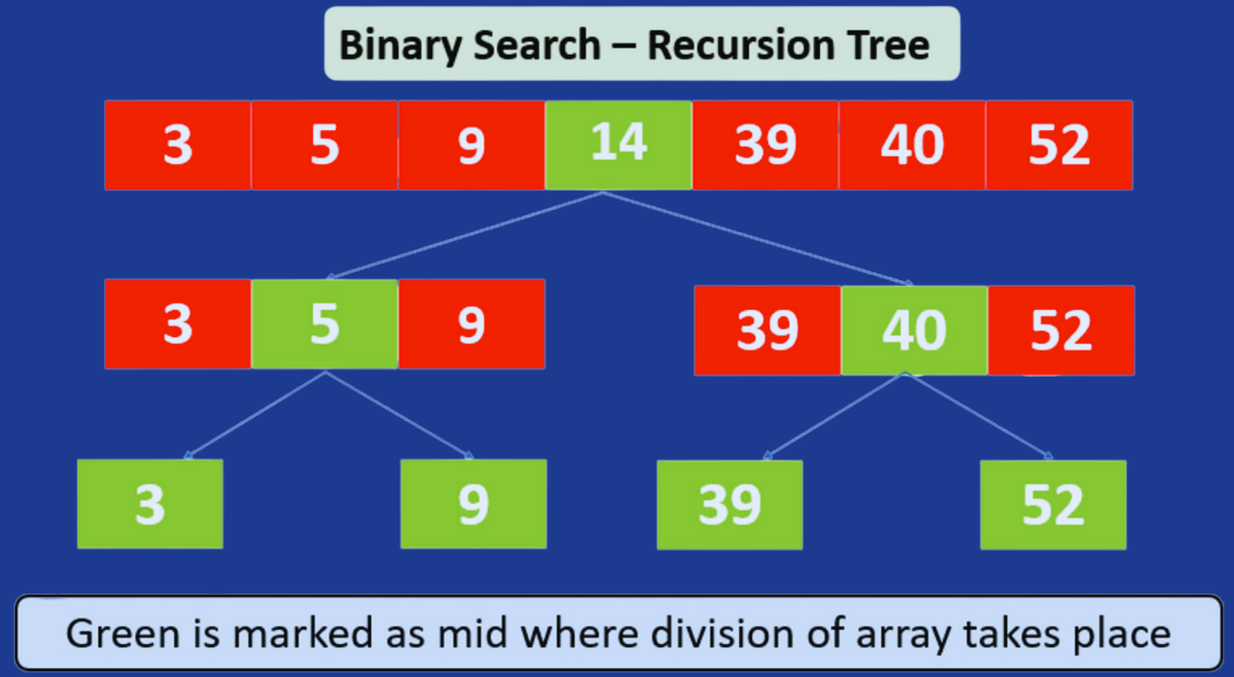
5/ APPLICATIONS OF DECREASE AND CONQUER

BINARY SEARCH

Binary Search is a fundamental algorithm used for searching in sorted arrays or lists. Imagine you have a sorted list of items and you're looking for a specific target value within that list. Binary Search efficiently locates the target value by repeatedly dividing the search interval in half.

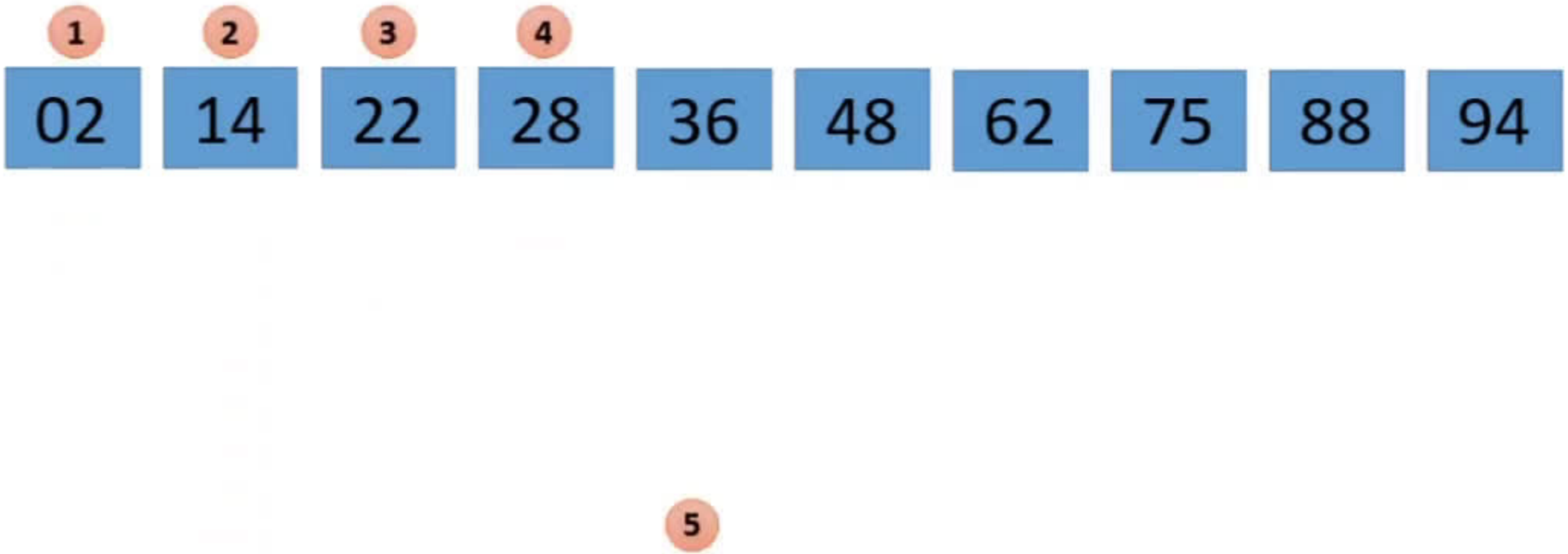
BINARY SEARCH

So at each step, the algorithm compares the target value with the middle element of the array. Based on this comparison, it eliminates half of the search space, proceeding to search in the remaining half. By continually halving the search interval, Binary Search quickly converges to the desired value or determines its absence.



Binary Search

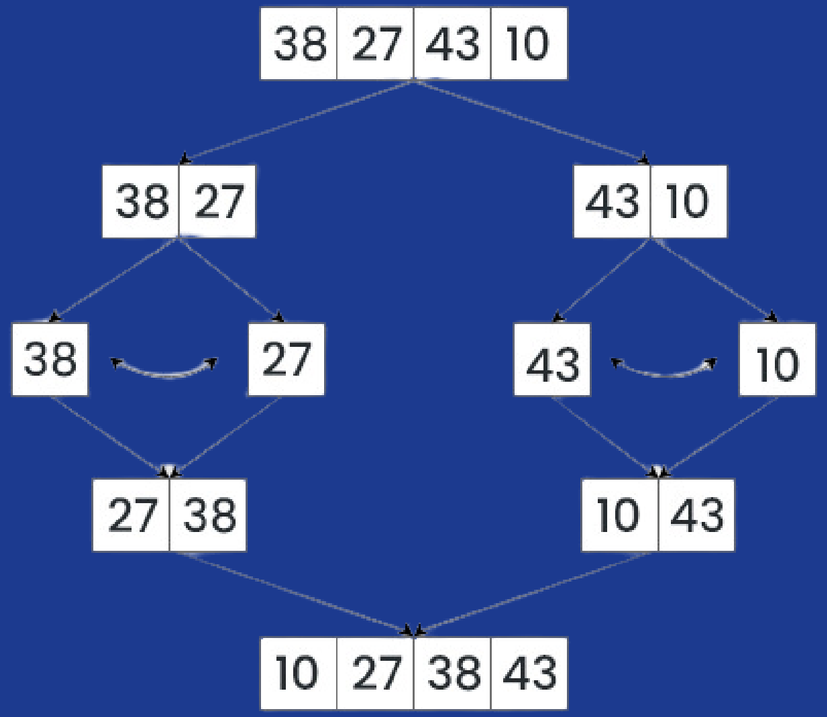
- **Let's Find 48,**



"**The iterative binary search algorithm** operates with a **time complexity of O(logn)**, indicating its efficiency in searching through large arrays. In contrast, the naive **(linear) search algorithm** exhibits a **time complexity of O(n)**, indicating a linear increase in search time with the size of the input array. Consequently, binary search outperforms linear search, particularly for large arrays, due to its logarithmic time complexity. Additionally, both **the recursive and iterative versions of the binary search algorithm** share the same time complexity of **O(logn)**, ensuring comparable efficiency in search operations."

MERGE SORT

The key idea behind Merge Sort lies in its two main operations: splitting and merging. During the splitting phase, the algorithm divides the array into halves until each sub-array contains only one element, effectively creating sorted sub-arrays. In the merging phase, the sorted sub-arrays are combined in a manner that preserves their order, resulting in a fully sorted array.



l = left index

r = right index

In this setup, the naive version of Merge Sort adopts an iterative method to split the array into sub-arrays. However, it employs a less efficient merging technique that entails creating a temporary list and iterating over the elements to merge them. Consequently, its time complexity tends to be larger compared to the recursive and iterative versions, which utilize more efficient merging techniques. Both the recursive and iterative versions achieve a time complexity of O(nlogn) due to their efficient merging strategies. In contrast, the naive version's time complexity may be higher due to its less efficient merging approach.

HERE'S A BREAKDOWN OF THE TIME COMPLEXITY

Merging Phase: In each iteration of the merging phase, all elements in the array are visited once. Since the number of iterations required for merging is O(logn) and in each iteration, we visit each element once, the total time complexity of the merging phase is O(n).

Splitting Phase: The array is repeatedly split into halves until each sub-array contains only one element. This splitting process requires O(logn) iterations, as the array is halved in size each time.

Therefore, the overall time complexity of the iterative Merge Sort algorithm is O(nlogn).

06/ADVANTAGES AND LIMITATIONS

ADVANTAGES

Simplifies Complex Problems: Decrease and Conquer breaks down complex problems into smaller, more manageable subproblems.

Facilitates Modular and Maintainable Code: It promotes modular code design, enhancing readability, reusability, and maintainability.

Enables Efficient Solutions: For problems with inherent substructure, it offers efficient solutions by reducing computational complexity.

ADVANTAGES

Simplifies Complex Problems: Decrease and Conquer breaks down complex problems into smaller, more manageable subproblems.

Facilitates Modular and Maintainable Code: It promotes modular code design, enhancing readability, reusability, and maintainability.

Enables Efficient Solutions: For problems with inherent substructure, it offers efficient solutions by reducing computational complexity.

LIMITATIONS AND CHALLENGES:

Recursion Depth Limitations: Recursive implementations may encounter stack overflow errors, especially for deeply nested calls.

Performance Issues for Large Problem Sizes: Efficiency may degrade for large problem instances due to recursion overhead or excessive iterations.

Not Universally Applicable: Not optimal for all problem types, particularly those lacking inherent substructure or with irregular patterns.

07/BEST PRACTICES AND TIPS

BEST PRACTICES AND TIPS

Identify Base Cases Accurately

Ensure precise identification of the smallest problem instances solvable directly without further decomposition.

Choose Appropriate Data Structures and Algorithms:

Select structures and algorithms considering time and space complexities, as well as problem nature.

Optimize Recursion to Minimize Overhead:

Streamline recursive implementations to reduce redundancy and ensure efficient termination.

Tips for Overcoming Common Challenges and Pitfalls:

- Rigorous testing and error handling to address bugs and edge cases early.
- Utilize debugging tools and online resources for assistance.

08/ CONCLUSION

CONCLUSION

Decrease and Conquer is a powerful algorithmic approach, breaking down complex problems into manageable parts for efficient solutions. Its principles, implementations, and applications underscore its significance in algorithm design. Let's harness its potential for innovation and problem-solving in our endeavors.

THANK YOU