

## Chapter 13

# Computational Approaches To Problem-Solving

*“The problem of computing is not about speed, it’s about understanding.”*

– Donald E Knuth

*“We build our machines in such a way that they carry out our instructions, and we in turn must carry out instructions given to us by our machines.”*

– Edsger W. Dijkstra

Computational approaches to problem-solving encompass a diverse range of methodologies that leverage computational power to address complex challenges across various domains. This chapter explores several fundamental strategies—brute force, divide-and-conquer, dynamic programming, greedy algorithms, and randomized approaches—each offering unique insights and techniques to tackle different classes of problems effectively.

The brute-force approach represents simplicity and exhaustive computation. It involves systematically checking every possible solution to find the optimal one, making it ideal for problems like cracking padlocks or guessing passwords. Despite its simplicity, brute-force methods can be computationally expensive, especially for problems with large solution spaces, leading to impractical execution times in real-world applications.

In contrast, the divide-and-conquer approach breaks down problems into smaller, more manageable sub-problems until they become simple enough to solve directly. The merge sort algorithm exemplifies this strategy by recursively dividing an array into halves, sorting them, and then merging them back together. This approach benefits from improved efficiency over brute-force methods by reducing the time complexity through systematic decomposition. However, it may incur additional overhead due to recursive function calls and memory requirements for storing sub-problems.

Dynamic programming focuses on solving problems by breaking them down

into overlapping sub-problems and storing the results to avoid redundant computations. Unlike divide-and-conquer, dynamic programming optimizes efficiency by memoizing intermediate results, significantly reducing computational complexity for problems with overlapping sub-problems. This approach highlights the trade-off between space and time complexity, making it particularly effective for optimization problems where solutions depend on prior computed results.

Greedy algorithms, such as maximizing the number of tasks completed within a limited time frame, make locally optimal choices at each step with the aim of reaching a global optimum. This approach is motivated by its simplicity and efficiency in finding quick solutions. However, greedy algorithms may overlook globally optimal solutions due to their myopic decision-making process, emphasizing immediate gains over long-term strategy.

Lastly, randomized approaches introduce randomness into problem-solving, offering probabilistic solutions to otherwise deterministic problems. Examples include scenarios like coupon collecting or hat-checking at a party, where outcomes depend on random chance rather than deterministic algorithms. Motivated by their ability to explore solution spaces unpredictably, randomized approaches provide insights into stochastic phenomena and offer innovative solutions in scenarios where exact solutions are impractical or unavailable.

This chapter will delve into each computational approach in detail, exploring their theoretical underpinnings, practical applications, advantages, and limitations. By understanding these methodologies, you will gain insights into selecting appropriate strategies for solving diverse computational problems effectively across various disciplines.

## 13.1 Brute-Force Approach to Problem Solving

*“To solve any problem, you need to start with a clear definition of the problem and then look at all possible solutions.”*

– John McCarthy

Many problems are addressed by exploring a vast number of possibilities. For instance, chess engines evaluate numerous move variations to determine the “best” positions. This method is known as brute force. Brute force algorithms take advantage of a computer’s speed, allowing us to rely less on sophisticated techniques. However, even with brute force, some level of creativity is often required. For example, a brute force solution might involve evaluating  $2^{40}$  options, but a more refined approach could potentially reduce this to  $2^{20}$ . Such a reduction can significantly decrease the computational time. The effectiveness of a brute force approach often hinges on how cleverly the problem is analyzed and optimized, making it crucial to explore different strategies to improve efficiency.

The brute-force approach is a fundamental method in problem-solving that involves systematically trying every possible solution to find the optimal one. This section explores the concept, applications, advantages, and limitations of the brute-force approach through various examples across different domains.

The brute-force approach, also known as exhaustive search, operates by checking all possible solutions systematically, without employing any sophisticated strategies to narrow down the search space. It ensures finding a solution if it exists within the predefined constraints but can be computationally intensive and impractical for problems with large solution spaces.

### Examples of Brute-Force Approach

#### 1. Padlock

Imagine you encounter a padlock with a four-digit numeric code. The brute-force approach would involve sequentially trying every possible combination from "0000" to "9999" until the correct code unlocks the padlock. Despite its simplicity and guaranteed success in finding the correct combination eventually, this method can be time-consuming, especially for longer or more complex codes.

2. **Password Guessing** In the realm of cybersecurity, brute-force attacks are used to crack passwords by systematically guessing every possible combination of characters until the correct password is identified. This approach is effective against weak passwords that are short or lack complexity. For instance, attacking a six-character password consisting of letters and digits would involve testing all 2.18 billion ( $36^6$ ) possible combinations until the correct one is identified.

#### 3. Cryptography: Cracking Codes

In cryptography, brute-force attacks are used to crack codes or encryption keys by systematically testing every possible combination until the correct one is found. For example, breaking a simple substitution cipher involves trying every possible shift in the alphabet until the plain-text message is deciphered.

#### 4. Sudoku Solving


Brute-force methods can be applied to solve puzzles like Sudoku by systematically filling in each cell with possible values and backtracking when contradictions arise. This method guarantees finding a solution but may require significant computational resources, especially for complex puzzles.

### 13.1.1 Characteristics of Brute-Force Solutions

1. **Exhaustive Search:** Every possible solution is examined without any optimization.
2. **Simplicity:** Easy to understand and implement.
3. **Inefficiency:** Often slow and resource-intensive due to the large number of possibilities.
4. **Guaranteed Solution:** If a solution exists, the brute-force method will eventually find it.

### 13.1.2 Solving Computational Problems Using Brute-force Approach Approach

To solve computational problems using the brute-force approach, one must systematically explore and evaluate all possible solutions to identify the correct answer. This involves defining the problem clearly, generating every potential candidate solution, and checking each one against the problem's criteria to determine its validity. While this method ensures that all possible solutions are considered, it often results in high computational costs and inefficiencies, especially for large or complex problems. Despite these limitations, the brute-force approach provides a straightforward and reliable way to solve problems by exhaustively searching the solution space, offering a foundation for understanding and improving more advanced algorithms.

 The brute-force approach is like searching for a needle in a haystack by sifting through each strand one by one; it is simple but can be overwhelmingly inefficient.

Let us look at some examples and see how we can apply the brute-force approach to solve a computational problem.

#### 13.1.2.1 Problem-1 (String Matching)

The brute-force string matching algorithm is a simple method for finding all occurrences of a pattern within a text. The idea is to slide the pattern over the text one character at a time and check if the pattern matches the substring of the text starting at the current position. Here is a step-by-step explanation:

1. **Start at the beginning of the text:** Begin by aligning the pattern with the first character of the text.
2. **Check for a match:** Compare the pattern with the substring of the text starting at the current position. If the substring matches the pattern, record the position.
3. **Move to the next position:** Shift the pattern one character to the right and repeat the comparison until you reach the end of the text.
4. **Finish:** Continue until all possible positions in the text have been checked.

This approach ensures that all possible starting positions in the text are considered, but it can be slow for large texts due to its time complexity.

Here is how you can implement the brute-force string-matching algorithm in Python:

```
def brute_force_string_match(text, pattern):
    n = len(text)          # Length of the text
```

```

m = len(pattern)    # Length of the pattern

for i in range(n - m + 1):
    substring = text[i:i + m]
    """ Loop over each possible starting index in the text,
    Extracting the substring of the text from the current
    position """

    # Compare the substring with the pattern
    if substring == pattern:
        print(f"Pattern found at index {i}")

# Example usage
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
brute_force_string_match(text, pattern)

```

**Explanation:**

1. **Function Definition:** The function `brute_force_string_match` takes two arguments: `text` and `pattern`.
2. **Length Calculation:** It calculates the lengths of both the `text` and `pattern` to determine how many possible starting positions there are.
3. **Loop Over Positions:** It uses a for loop to slide the pattern across the text. The loop runs from `0` to `n - m + 1`, where `n` is the length of the `text` and `m` is the length of the `pattern`.
4. **Substring Extraction:** At each position `i`, the code extracts a substring from the text that has the same length as the pattern (`text[i:i + m]`).
5. **Comparison:** It then compares this substring with the pattern. If they match, it prints the starting index where the pattern was found.
6. **Example Usage:** The example shows how to call the function with a sample text and pattern. In this case, the function prints the indices where the pattern occurs within the text.

This implementation is straightforward and guarantees finding all occurrences of the pattern, but it may not be efficient for large texts or patterns due to its  $((n - m + 1) * m)$  time complexity.

Given the example usage in our `brute_force_string_match` function:

```

text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
brute_force_string_match(text, pattern)

```

Here is what the output of the function will be:

```

Pattern found at index 10

```

The function searches through the **text** and finds the **pattern** starting at index **10**. In this case, "**ABABCABAB**" begins at position **10** in the **text**, so that is where the function prints the message indicating the pattern's location.

The function does not find the pattern at any other starting position in the provided text, so only this single index is printed.

### 13.1.2.2 Problem-2 (Subset Sum Problem)

The Subset Sum Problem involves determining if there exists a subset of a given set of numbers that sums up to a specified target value. The brute-force approach to solve this problem involves generating all possible subsets of the set and checking if the sum of any subset equals the target value.

Here is how the brute-force approach works:

1. **Generate subsets:** Iterate over all possible subsets of the given set of numbers.
2. **Calculate sums:** For each subset, calculate the sum of its elements.
3. **Check target:** Compare the sum of each subset with the target value.
4. **Return result:** If a subset's sum matches the target, return that subset. Otherwise, conclude that no such subset exists.

This method guarantees finding a solution if one exists but can be inefficient for large sets due to its exponential time complexity.

Here is a Python code to implement the brute-force approach for the Subset Sum Problem:

```

def subset_sum_brute_force(nums, target):
    n = len(nums)

    # Loop over all possible subsets

    for i in range(1 << n): # There are 2^n subsets
        subset = [nums[j] for j in range(n) if (i & (1 << j))]
        if sum(subset) == target:
            return subset

    return None

```

```
# Example usage
nums = [3, 34, 4, 12, 5, 2]
target = 9
result = subset_sum_brute_force(nums, target)
if result:
    print(f"Subset with target sum {target} found: {result}")
else:
    print("No subset with the target sum found.")
```

**Explanation:**

1. **Function Definition:** `subset_sum_brute_force` takes a list of numbers (**nums**) and a target sum (**target**).
2. **Subset Generation:** The loop `for i in range(1 « n)` iterates over all possible subsets. Here, `1 « n` equals  $2^n$ , which is the total number of subsets for **n** elements. Each subset is generated using a bitmask approach: for each bit in the integer **i**, if it is set, the corresponding element is included in the subset.
3. **Subset Construction:** The subset is constructed by including elements where the corresponding bit in **i** is set (`(i & (1 « j))`).
4. **Sum Calculation:** For each subset, the sum of its elements is calculated using `sum(subset)`.
5. **Target Check:** If the sum of the subset equals the target, the subset is returned.
6. **Return Result:** If no subset matches the target sum, the function returns **None**.
7. **Example Usage:** The example demonstrates finding a subset that sums up to **9** in the list `[3, 34, 4, 12, 5, 2]`. The output will either show the subset that matches the target or indicate that no such subset was found.

This brute-force approach is straightforward and guarantees finding a solution if one exists, but may not be efficient for large sets due to its exponential time complexity.

Given the example usage in our `subset_sum_brute_force` function:

```
nums = [3, 34, 4, 12, 5, 2]
target = 9
result = subset_sum_brute_force(nums, target)
if result:
    print(f"Subset with target sum {target} found: {result}")
else:
    print("No subset with the target sum found.")
```

Here is what the output of the function will be:

Subset with target sum 9 found: [3, 4, 2]

The function generates all possible subsets of the list **nums** and checks if any of them sum up to the target value **9**.

- **Subset Generation:** The function iterates through all possible subsets. For each subset, it calculates the sum and checks if it matches the target.
- **Subset Found:** In this case, the subset [3, 4, 2] sums up to 9, which matches the target value. Therefore, this subset is returned and printed.

If no such subset were found, the function would print "No subset with the target sum found."

### 13.1.2.3 Problem-3 (Sudoku Solver)

The Sudoku Solver using the brute-force approach is a method to solve a Sudoku puzzle by trying every possible number in each empty cell until the puzzle is solved. The brute-force algorithm systematically fills in each cell with numbers from 1 to 9 and checks if the puzzle remains valid after each placement. If a placement leads to a valid state, the algorithm proceeds to the next empty cell. If a placement leads to a contradiction, the algorithm backtracks and tries the next number.

Here is a step-by-step explanation:

1. **Find an Empty Cell:** Locate the first empty cell in the Sudoku grid.
2. **Try Numbers:** Attempt to place each number from 1 to 9 in the empty cell.
3. **Check Validity:** Verify that placing the number does not violate Sudoku rules:
  - No repeated numbers in the same row.
  - No repeated numbers in the same column.
  - No repeated numbers in the same 3x3 sub-grid.
4. **Move to Next Cell:** If the placement is valid, proceed to the next empty cell.
5. **Backtrack if Necessary:** If a placement leads to an invalid state later, undo the placement (backtrack) and try the next number.
6. **Complete:** Continue until the Sudoku puzzle is fully solved or all possibilities are exhausted.



The brute-force approach guarantees finding a solution if one exists, but it can be inefficient for larger puzzles due to its exhaustive nature.

Here is the Python code for solving a Sudoku puzzle using the brute-force approach with recursive backtracking::

```
def is_valid(board, row, col, num):
    # Check if num is not repeated in the row

    if num in board[row]:
        return False

    # Check if num is not repeated in the column

    if num in (board[i][col] for i in range(9)):
        return False

    # Check if num is not repeated in the 3x3 sub-grid

    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(start_row, start_row + 3):
        for j in range(start_col, start_col + 3):
            if board[i][j] == num:
                return False

    return True

def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0: # Find an empty cell
                for num in range(1, 10): # Try all numbers
                    # from 1 to 9
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        # Place the number
                        if solve_sudoku(board):
                            return True
                        board[row][col] = 0 # Backtrack if
                    needed
                return False # Trigger backtracking

    return True # Puzzle solved

# Example usage
sudoku_board = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
```

```
[6, 0, 0, 1, 9, 5, 0, 0, 0],
[0, 9, 8, 0, 0, 0, 0, 6, 0],
[8, 0, 0, 0, 6, 0, 0, 0, 3],
[4, 0, 0, 8, 0, 3, 0, 0, 1],
[7, 0, 0, 0, 2, 0, 0, 0, 6],
[0, 6, 0, 0, 0, 0, 2, 8, 0],
[0, 0, 0, 4, 1, 9, 0, 0, 5],
[0, 0, 0, 0, 8, 0, 0, 7, 9]
]

if solve_sudoku(sudoku_board):
    for row in sudoku_board:
        print(row)
else:
    print("No solution exists.")
```

**Explanation:**

1. **is\_valid Function:** This function checks if placing a number in a specific cell is valid according to Sudoku rules:
  - **Row Check:** Ensures the number is not already present in the same row.
  - **Column Check:** Ensures the number is not already present in the same column.
  - **Sub-grid Check:** Ensures the number is not already present in the 3x3 sub-grid.
2. **solve\_sudoku Function:**
  - **Find Empty Cell:** Iterates through the grid to locate an empty cell (0).
  - **Try Numbers:** For each empty cell, try placing numbers from 1 to 9.
  - **Check Validity:** Uses **is\_valid** to check if the placement is valid.
  - **Recursive Call:** Recursively attempts to solve the rest of the board with the current placement.
  - **Backtrack:** If no valid number can be placed, reset the cell and try the next number.
  - **Completion:** If all cells are filled validly, returns **True**. If no cells are left to fill, returns **True** indicating the board is solved.
3. **Example Usage:** Demonstrates solving a Sudoku puzzle with a given **sudoku\_board**. If the puzzle is solved, the board is printed row by row. If no solution exists, a message is displayed.

This brute-force algorithm ensures a solution if one exists but can be slow due to its exhaustive search approach.

Here is what the output would look like if the provided `solve_sudoku` function successfully solves the given Sudoku puzzle:

```
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```


#### Explanation of the Output:

1. **Completed Sudoku Grid:** Each row shows a valid configuration where all rows, columns, and 3x3 sub-grids contain the numbers 1 through 9 without repetition.
2. **Successful Solution:** The `solve_sudoku` function filled all empty cells (originally 0 values) with valid numbers, resulting in a fully completed Sudoku board.

If the `solve_sudoku` function did not find a solution, it would print:

```
No solution exists.
```

This approach guarantees to find a solution if one exists but might be slow for more complex or larger Sudoku puzzles due to its exhaustive nature.

 Brute-force methods offer a straightforward path to solving problems by exploring every possible solution, but they often become impractical as the problem size grows.

### 13.1.3 Advantages and Limitations of Brute-Force Approach


#### Advantages of Brute-Force Approach

1. **Guaranteed Solution:** Brute-force methods ensure finding a solution if one exists within the predefined constraints.
2. **Simplicity:** The approach is straightforward to implement and understand, requiring minimal algorithmic complexity.

3. **Versatility:** Applicable across various domains where an exhaustive search is feasible, such as puzzle solving, cryptography, and optimization problems.


### Limitations of Brute-Force Approach

1. **Computational Intensity:** It can be highly resource-intensive, especially for problems with large solution spaces or complex constraints.
2. **Time Complexity:** Depending on the problem size, brute-force approaches may require impractically long execution times to find solutions.
3. **Scalability Issues:** In scenarios with exponentially growing solution spaces, brute-force methods may become impractical or infeasible to execute within reasonable time constraints.

 Simplicity is the hallmark of brute-force algorithms; they operate without complex strategies but may suffer from exponential growth in computational demands.

### 13.1.4 Optimizing Brute-force Solutions

- **Pruning:** Eliminate certain candidates early if they cannot possibly be a solution. For example, in a search tree, cutting off branches does not lead to feasible solutions.
- **Heuristics:** Use rules of thumb to guide the search and reduce the number of candidates.
- **Divide and Conquer:** Break the problem into smaller, more manageable parts, solve each part individually, and combine the results.
- **Dynamic Programming:** Store the results of subproblems to avoid redundant computations.

 While brute-force approaches can serve as a baseline for evaluating other algorithms, their inefficiency limits their practical use to small-scale problems or as a verification tool.

The brute-force approach is a fundamental but often inefficient problem-solving technique. While it guarantees finding a solution if one exists, its practical use is limited by computational constraints. Understanding brute-force methods is essential for developing more sophisticated algorithms and optimizing problem-solving strategies. This section provides an in-depth exploration of the brute-force approach for solving computational problems, highlighting its simplicity, applications, limitations, and potential optimizations.

## 13.2 Divide-and-conquer Approach to Problem Solving

*The process of breaking down a complex problem into simpler sub-problems is not only a fundamental programming technique but also a powerful strategy for managing complexity.”*

– Donald E Knuth

To illustrate the divide-and-conquer approach, imagine a classroom scenario where students are asked to organize a large number of books in a library. The problem of categorizing and shelving thousands of books can seem difficult at first. By applying divide-and-conquer principles, the task can be simplified significantly. The students can start by dividing the books into smaller groups based on genres, such as fiction, non-fiction, and reference. Each genre can then be further subdivided into categories like science fiction, historical novels, and biographies. Finally, within each category, the books can be organized alphabetically by author. This method of dividing the problem into manageable parts, solving each part, and then combining the results effectively demonstrates how divide-and-conquer can make complex tasks more approachable.

In the realm of project management, divide-and-conquer strategies are essential for handling large projects. For example, consider the construction of a high-rise building. The project is divided into smaller tasks, such as foundation work, structural framework, electrical installations, and interior finishes. Each task is managed by different teams or contractors specialized in that area. By breaking the project into these distinct components, project managers can ensure that each part is completed efficiently and effectively. This modular approach allows for parallel progress, timely completion, and integration of the individual tasks to achieve the final goal of constructing the building.

In software development, the divide-and-conquer approach is frequently employed in designing complex systems and applications. For instance, consider developing a comprehensive e-commerce platform. The platform is divided into various functional modules, such as user authentication, product catalog, shopping cart, and payment processing. Each module is developed and tested independently, allowing developers to focus on specific aspects of the system. Once all modules are completed, they are integrated to form a cohesive application. This method ensures that the development process is manageable and that each component functions correctly before being combined into the final product.

In healthcare, divide-and-conquer strategies are applied in diagnostic processes and treatment plans. For example, when diagnosing a complex medical condition, doctors might first divide the patient's symptoms into different categories, such as neurological, cardiovascular, and respiratory. Each category is investigated separately using targeted tests and consultations with specialists. The results from these investigations are then combined to form a comprehensive diagnosis and treatment plan. This approach helps in managing the complexity of medical diagnoses and ensures a thorough and accurate evaluation of the

patient's health.

Finally, consider the field of logistics and supply chain management, where divide-and-conquer techniques are used to optimize the distribution of goods. For example, a company managing the supply chain for a large retailer might divide the supply chain into regional distribution centers. Each center handles a specific geographic area and manages local inventory, transportation, and delivery. By decentralizing the management of the supply chain into smaller, regional units, the company can improve efficiency, reduce costs, and enhance service levels. The results from each distribution center are then integrated to ensure a seamless supply chain operation across all regions.

As illustrated in these examples, the divide-and-conquer approach is also a fundamental computational problem-solving technique used to solve problems by breaking them down into smaller, more manageable sub-problems similar to the original problem. The basic idea is to divide the problem into smaller sub-problems, solve each sub-problem independently, and then combine their solutions to solve the original problem. This method is particularly effective for problems that exhibit recursive structure and can be broken into similar sub-problems.

#### Key Steps of Divide-and-Conquer:

1. **Divide:** Split the original problem into smaller sub-problems that are easier to solve. The sub-problems should be similar to the original problem.

Consider the task of organizing a large set of files into a well-structured directory system. The first step involves breaking down this problem into smaller, more manageable subproblems. For instance, you might divide the files by their type (e.g., documents, images, videos) or by their project affiliation. Each subset of files is then considered a subproblem, which is more straightforward to organize than the entire set of files. The key is to ensure that each subset is similar to the original problem but simpler to handle individually.

2. **Conquer:** Solve the smaller sub-problems. If the sub-problems are small enough, solve them directly. Otherwise, apply the divide-and-conquer approach recursively to these sub-problems

Once the files are divided into smaller subsets, each subset is organized recursively. For example, you could sort documents into subcategories such as reports, presentations, and spreadsheets. Each of these categories might be further divided into subfolders based on date or project. This recursive approach allows you to systematically manage and categorize each subset. For very small subsets, such as a single folder with a few files, a direct solution is applied without further division, making the problem-solving process more manageable.

3. **Combine:** Combine the solutions of the sub-problems to form the solution to the original problem.

After each subset of files is organized, the final step is to combine these organized subsets into the overall directory system. This involves merging the categorized folders into a hierarchical structure that reflects the original file organization plan. The result is a complete and well-structured directory system that maintains the overall organization and makes it easy to locate and manage files. The combination of the organized subsets ensures that the entire file system is coherent and functional, achieving the goal of efficient file organization.

By applying these steps, the complex task of organizing a large set of files is broken down into manageable steps, leveraging recursion to handle each subset and integrating the results into a comprehensive directory system. This section will explore the fundamental principles of the divide-and-conquer strategy, its applications, and its advantages and disadvantages.

### 13.2.1 Principles of Divide-and-Conquer

#### 13.2.1.1 Divide


The first step involves breaking down the problem into smaller subproblems. This division should be done so that the subproblems are similar to the original problem. The key is to ensure that each subproblem is easier to solve than the original.

#### 13.2.1.2 Conquer

Once the problem is divided, each subproblem is solved recursively. This step leverages the power of recursion, making the problem-solving process more manageable. For very small subproblems, a direct solution is applied without further division.

#### 13.2.1.3 Combine

The final step involves combining the solutions of the subproblems to form the solution to the original problem. This step often requires merging results in a manner that maintains the problem's overall structure and requirements.

 Divide-and-conquer breaks a problem into smaller, manageable subproblems, solving each independently and combining their solutions to address the original problem.

Let us walk through an example to get an idea about how the principle of divide-and-conquer is applied to solve computational problems.

#### Example: Merge Sort Algorithm

Merge Sort is a classic example of the divide-and-conquer strategy used for sorting an array of elements. It operates by recursively breaking down the array into progressively smaller sections. The core idea is to split the array into two halves, sort each half, and then merge them back together. This process continues until the array is divided into individual elements, which are inherently sorted.

The merging process relies on a straightforward principle: when combining two sorted halves, the smallest value of the entire array must be the smallest value from either of the two halves. By iteratively comparing the smallest elements from each half and appending the smaller one to the sorted array, we efficiently merge the halves into a fully sorted array. This approach is not only intuitive but also simplifies the coding of the recursive splits and the merging procedure. Here is how it works:

1. **Divide:** Split the array into two halves.
2. **Conquer:** Recursively sort both halves.
3. **Combine:** Merge the two sorted halves to produce the sorted array.

To get an idea of how the Merge sort works, let us visualize the working of the algorithm. Visualizing the Merge Sort algorithm helps to understand how the divide-and-conquer approach works by breaking down the array into smaller parts and then merging them back together. Here's a step-by-step visualization of Merge Sort:

#### Visualization Steps:

1. **Divide:** The array is recursively divided into two halves until each sub-array contains a single element.
2. **Merge:** The sub-arrays are then merged in a sorted order.

Let us use an example array: [38, 27, 43, 3, 9, 82, 10].

#### Step 1: Divide the Array

1. **Initial Array:** [38, 27, 43, 3, 9, 82, 10]
2. **Divide into Halves:**
  - Left Half: [38, 27, 43]
  - Right Half: [3, 9, 82, 10]
3. **Further Divide:**
  - Left Half: [38, 27, 43] becomes:
    - \* [38] and [27, 43]
    - \* [27, 43] becomes:
      - [27] and [43]



- Right Half: **[3, 9, 82, 10]** becomes:
  - \* **[3, 9]** and **[82, 10]**
  - \* **[3, 9]** becomes:
    - **[3]** and **[9]**
  - \* **[82, 10]** becomes:
    - **[82]** and **[10]**

### Step 2: Merge the Arrays

1. **Merge Smaller Arrays:**
  - **[27]** and **[43]** are merged to form **[27, 43]**
  - **[3]** and **[9]** are merged to form **[3, 9]**
  - **[82]** and **[10]** are merged to form **[10, 82]**
2. **Merge Larger Arrays:**
  - **[38]** and **[27, 43]** are merged to form **[27, 38, 43]**
  - **[3, 9]** and **[10, 82]** are merged to form **[3, 9, 10, 82]**
3. **Final Merge:**
  - **[27, 38, 43]** and **[3, 9, 10, 82]** are merged to form the sorted array **[3, 9, 10, 27, 38, 43, 82]**

This is how the merge sort algorithm works, breaking down the problem into smaller subproblems, solving each independently, and combining the results to get the final sorted array.

To visualize this process, here is a diagram representing the Merge Sort:

#### Explanation:

1. **Initial Split:** The array is divided into smaller chunks recursively.
2. **Recursive Sorting:** Each chunk is sorted individually.
3. **Merging:** The sorted chunks are merged back together in sorted order.

By following this visualization, you can see how the divide-and-conquer approach efficiently breaks down the problem and then builds up the solution step-by-step. This method ensures that each element is placed in its correct position in the final sorted array.

Here is a detailed explanation in English of the merge sort algorithm along with the merge function:

#### 1. Function mergeSort

- Check if the array has one or zero elements. If true, return the array as it is already sorted.
- Otherwise, find the middle index of the array.
- Split the array into two halves: from the beginning to the middle and from the middle to the end.
- Recursively apply mergeSort to the first half and the second half.

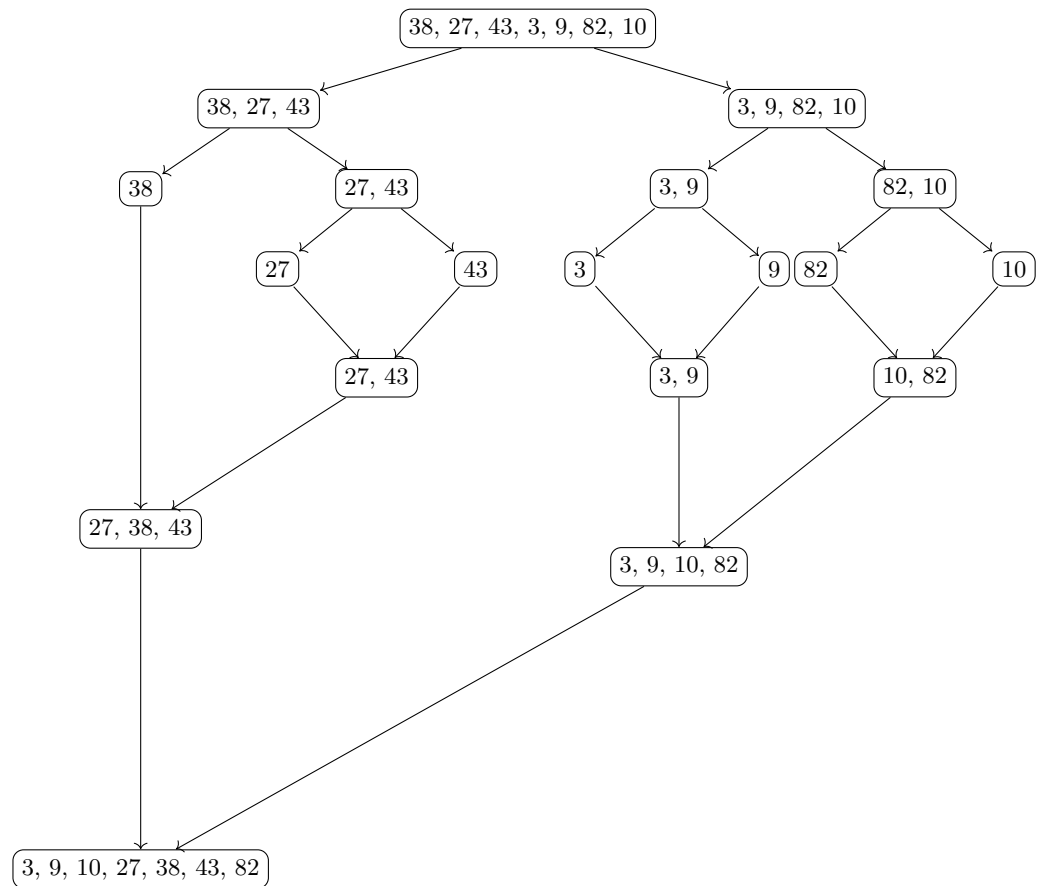


Figure 13.1: The recursion tree given when performing a recursive split of the array  $[38, 27, 43, 3, 9, 82, 10]$ .

- Merge the two sorted halves using the merge function.
- Return the merged and sorted array.

## 2. Function merge

- Create an empty list called **sorted\_arr** to store the sorted elements.
- While both halves have elements:
  - Compare the first element of the left half with the first element of the right half.
  - Remove the smaller element and append it to the **sorted\_arr** list.
- If the left half still has elements, append them all to the **sorted\_arr** list.
- If the right half still has elements, append them all to the **sorted\_arr** list.
- Return the **sorted\_arr** list, which now contains the sorted elements from both halves.

Here is a Python implementation of Merge Sort

```
def merge_sort(arr):
    """ Sorts an array using the merge sort algorithm. """
    if len(arr) <= 1:
        return arr

    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    # Combine the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
    """ Merges two sorted arrays into one sorted array. """
    sorted_arr = []
    i = j = 0

    # Merge the two arrays
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1
```

```

        j += 1

    # Add remaining elements (if any)
    while i < len(left):
        sorted_arr.append(left[i])
        i += 1
    while j < len(right):
        sorted_arr.append(right[j])
        j += 1

    return sorted_arr

# Example usage
array = [38, 27, 43, 3, 9, 82, 10]
sorted_array = merge_sort(array)
print("Sorted array:", sorted_array)

```

**Explanation:****1. merge\_sort Function:**

- If the array has one or zero elements, it's already sorted.
- The array is divided into two halves recursively until the base case is reached (arrays of size one).
- The sorted halves are combined using the merge function.

**2. merge Function:**

- Takes two sorted arrays and merges them into one sorted array.
- Compares elements from both arrays and appends the smaller element to the result array.
- After one array is exhausted, append any remaining elements from the other array.

### 13.2.2 Solving Computational Problems Using Divide and Conquer Approach

Let us see how we can apply the principles of the divide-and-conquer approach to solve computational problems.

#### 13.2.2.1 Problem-1 (Finding the Maximum Element in an Array)

Given an array of integers, find the maximum value in the array.

**Step-by-Step Solution****1. Initial Setup:**

- Begin with the entire array and determine the range to process. Initially, this range includes the entire array from the first element to the last element.

## 2. Divide

- If the array contains more than one element, split it into two approximately equal halves. This splitting continues recursively until each subarray has only one element.

## 3. Conquer:

- For subarrays with only one element, that element is trivially the maximum for that subarray.
- For larger subarrays, recursively apply the same process to each half of the subarray.

## 4. Combine:

- After finding the maximum element in each of the smaller subarrays, combine the results by comparing the maximum values from each half. Return the largest of these values as the maximum for the original array.

## Python Implementation

Here is how to implement this algorithm in Python

```
def find_max(arr, left, right):
    # Base case: If the array segment has only one element
    if left == right:
        return arr[left]

    # Divide: Find the middle point of the current segment
    mid = (left + right) // 2

    """ Conquer: Recursively find the maximum in the left and
    right halves """

    max_left = find_max(arr, left, mid) # Maximum in the left
    half
    max_right = find_max(arr, mid + 1, right) # Maximum in the
    right half

    # Combine: Return the maximum of the two halves

    return max(max_left, max_right)
```

```
# Example usage
array = [3, 6, 2, 8, 7, 5, 1]
result = find_max(array, 0, len(array) - 1)
print("Maximum element:", result) # Output: Maximum element: 8
```

### Step-by-Step Explanation:

#### 1. Initial Setup:

- The **find\_max** function is called with the entire array and indices that cover the whole array. For example, **find\_max(array, 0, len(array) - 1)**.

#### 2. Divide:

- Calculate the middle index **mid** of the current array segment. For the array **[3, 6, 2, 8, 7, 5, 1]**, **mid** would be  $(0 + 6) // 2 = 3$ .
- Split the array into two halves based on this **mid** index
  - Left half: **[3, 6, 2, 8]**
  - Right half: **[7, 5, 1]**

#### 3. Conquer:

- Recursively apply the **find\_max** function to the left half **[3, 6, 2, 8]**:
  - Split into **[3, 6]** and **[2, 8]**
  - Further split **[3, 6]** into **[3]** and **[6]**, finding **3** and **6**, respectively.
  - Combine these to get the maximum **6**.
  - Similarly, split **[2, 8]** into **[2]** and **[8]**, finding **2** and **8**, respectively.
  - Combine these to get the maximum **8**.
  - Combine **6** and **8** from the two halves to get **8**.
- Apply the same process to the right half **[7, 5, 1]**:
  - Split into **[7]** and **[5, 1]**
  - Further split **[5, 1]** into **[5]** and **[1]**, finding **5** and **1**, respectively.
  - Combine these to get the maximum **5**.
  - Combine **7** and **5** from the two halves to get **7**.

#### 4. Combine:

- Finally, compare the maximums obtained from the left half (**8**) and the right half (**7**).
- Return the larger value, which is **8**.

This method efficiently finds the maximum element in the array by recursively dividing the problem, solving the subproblems, and combining the results.

**13.2.2.2 Problem-2 (Finding the Maximum Subarray Sum)**

Given an array of integers, which include both positive and negative numbers, find the contiguous subarray that has the maximum sum.

**Step-by-Step Solution****1. Initial Setup:**

- If the array contains only one element, that element is the maximum subarray sum.

**2. Divide:**

- Divide the array into two approximately equal halves. This involves finding the middle index of the array and separating the array into a left half and a right half.

**3. Conquer:**

- Recursively find the maximum subarray sum for:
  - The left half of the array.
  - The right half of the array.
  - The subarray that crosses the middle boundary between the two halves.

**4. Combine:**

- Combine the results from the three areas:
  - The maximum subarray sum in the left half.
  - The maximum subarray sum in the right half.
  - The maximum subarray sum that crosses the middle.
- Return the largest of these three values.

**Python Implementation**

Here is the Python code implementing this algorithm

```
def max_crossing_sum(arr, left, mid, right):
    # Find maximum sum of subarray crossing the middle point

    # Start from mid and move left

    left_sum = float('-inf')
    sum_left = 0
    for i in range(mid, left - 1, -1):
        sum_left += arr[i]
        if sum_left > left_sum:
```

```

        left_sum = sum_left

        # Start from mid+1 and move right

        right_sum = float('-inf')
        sum_right = 0
        for i in range(mid + 1, right + 1):
            sum_right += arr[i]
        if sum_right > right_sum:
            right_sum = sum_right

        """ Return the maximum sum of the subarray that crosses the
        mid """

        return left_sum + right_sum

def max_subarray_sum(arr, left, right):
    # Base case: only one element
    if left == right:
        return arr[left]

    # Divide the array into two halves
    mid = (left + right) // 2

    # Find maximum subarray sum in the left half
    left_max = max_subarray_sum(arr, left, mid)

    # Find maximum subarray sum in the right half
    right_max = max_subarray_sum(arr, mid + 1, right)

    # Find maximum subarray sum crossing the middle
    cross_max = max_crossing_sum(arr, left, mid, right)

    # Return the maximum of the three results
    return max(left_max, right_max, cross_max)

# Example usage
array = [2, 3, -4, 5, -1, 2, 3, -2, 4]
result = max_subarray_sum(array, 0, len(array) - 1)
print("Maximum subarray sum:", result) # Output: Maximum
subarray sum: 9

```

### Step-by-Step Explanation:

#### 1. Initial Setup:

- If the array segment contains only one element (base case), return



that element as the maximum subarray sum.

**2. Divide:**

- Calculate the middle index: **mid** = (**left** + **right**) // 2.
- Divide the array into two subarrays:
  - Left subarray from **left** to **mid**.
  - Right subarray from **mid** + 1 to **right**.

**3. Conquer:**

- **Find Maximum Subarray Sum in the Left Half:**
  - Recursively call **max\_subarray\_sum** on the left half.
- **Find Maximum Subarray Sum in the Right Half:**
  - Recursively call **max\_subarray\_sum** on the right half.
- **Find Maximum Subarray Sum Crossing the Middle:**
  - Call **max\_crossing\_sum** to find the maximum subarray sum that crosses the midpoint of the array. This involves calculating the maximum sum of the subarray that ends in the left half and starts in the right half.

**4. Combine:**

- The result is the maximum of:
  - The maximum sum found in the left half.
  - The maximum sum found in the right half.
  - The maximum sum of the subarray crossing the middle.

**Example Walkthrough:**

**Array:** [2, 3, -4, 5, -1, 2, 3, -2, 4]

**1. Initial Setup:**

- The array is divided into subarrays until base cases with single elements are reached.

**2. Divide:**

- For the array [2, 3, -4, 5, -1, 2, 3, -2, 4], **mid** = 4, so divide into:
  - Left half: [2, 3, -4, 5, -1]
  - Right half: [2, 3, -2, 4]

**3. Conquer:**

- **Left Half** [2, 3, -4, 5, -1]:
  - Further divide into [2, 3] and [-4, 5, -1].

- Find maximum subarray sums for these segments and combine them.
- **Right Half [2, 3, -2, 4]:**
  - Further divide into [2, 3] and [-2, 4].
  - Find maximum subarray sums for these segments and combine them.
- **Crossing Subarray:**
  - Calculate the maximum sum of subarrays crossing the middle index for both halves.

#### 4. Combine:

- Compare the maximum subarray sums from the left half, right half, and crossing subarray to get the overall maximum.

#### Explanation of Crossing Subarray

Consider the array: [2, 3, -4, 5, -1, 2, 3, -2, 4]

Let us find the maximum subarray sum that crosses the midpoint of the array.

#### Step-by-Step Explanation:

##### 1. Divide the Array:

- Suppose we are working with the whole array and the midpoint is calculated to be 4. So we divide the array into two halves:
  - Left half: [2, 3, -4, 5, -1]
  - Right half: [2, 3, -2, 4]
- Our goal is to find the maximum sum of subarrays that might cross this midpoint between the two halves.

##### 2. Find the Maximum Crossing Subarray:

- To find the maximum subarray sum that crosses the midpoint, we need to consider two parts:
  - The part of the subarray that extends from the midpoint to the left end
  - The part of the subarray that extends from the midpoint to the right end.
- **Calculate the Maximum Sum of the Left Part:**
  - Start from the midpoint and extend leftward.
  - Track the maximum sum while extending leftward from the midpoint.

- In the example, the midpoint is **4**, so we start from index **4** and move left.

Array segment: **[-1, 5, -4, 3, 2]**

- Compute maximum subarray sum ending at the midpoint:

- \* Start from **-1**, sum is **-1**.
- \* Extend to include **5**: sum becomes **4**.
- \* Extend to include **-4**: sum becomes **0**.
- \* Extend to include **3**: sum becomes **3**.
- \* Extend to include **2**: sum becomes **5**.

The maximum subarray sum ending at the midpoint (and extending leftward) is **5**.

- **Calculate the Maximum Sum of the Right Part:**

- Start from the midpoint + 1 and extend rightward.
- Track the maximum sum while extending rightward from the midpoint

Array segment: **[2, 3, -2, 4]**

- Compute maximum subarray sum starting from the midpoint + 1:

- \* Start from **2**, sum is **2**.
- \* Extend to include **3**: sum becomes **5**.
- \* Extend to include **-2**: sum becomes **3**.
- \* Extend to include **4**: sum becomes **7**.

The maximum subarray sum starting at midpoint + 1 (and extending rightward) is **7**.

- **Combine the Results:**

- The maximum sum of the subarray that crosses the midpoint is the sum of the maximum sum of the left part and the maximum sum of the right part.
- From the above calculations, the maximum crossing sum is **5 (left) + 7 (right) = 12**.

The crossing subarray that yields the maximum sum spans the midpoint and consists of elements from both the left and right parts. By combining the best sums of subarrays extending from the midpoint, we get the total maximum crossing sum.

This approach efficiently computes the maximum subarray sum using the divide and conquer strategy, which can be particularly useful for larger arrays due to its manageable number of comparisons.

 By recursively dividing a complex problem into simpler components,

the divide-and-conquer approach transforms large-scale challenges into solvable tasks with a clear, structured strategy.

### 13.2.3 Advantages and Disadvantages of Divide and Conquer Approach


#### Advantages of Divide and Conquer Approach

1. **Simplicity in Problem Solving:** By breaking a problem into smaller subproblems, each subproblem is simpler to understand and solve, making the overall problem more manageable.
2. **Efficiency:** Many divide-and-conquer algorithms, such as merge sort and quicksort, have optimal or near-optimal time complexities. These algorithms often have lower time complexities compared to iterative approaches.
3. **Modularity:** Divide-and-conquer promotes a modular approach to problem-solving, where each subproblem can be handled by a separate function or module. This makes the code easier to maintain and extend.
4. **Reduction in Complexity:** By dividing the problem, the overall complexity is reduced, and solving smaller subproblems can lead to simpler and more efficient solutions.
5. **Parallelism:** The divide-and-conquer approach can easily be parallelized because the subproblems can be solved independently and simultaneously on different processors, leading to potential performance improvements.
6. **Better Use of Memory:** Some divide-and-conquer algorithms use memory more efficiently. For example, the merge sort algorithm works well with large data sets that do not fit into memory, as it can process subsets of data in chunks.

#### Disadvantages of Divide and Conquer Approach

1. **Overhead of Recursive Calls:** The recursive nature can lead to significant overhead due to function calls and maintaining the call stack. This can be a problem for algorithms with deep recursion or large subproblem sizes.
2. **Increased Memory Usage:** Divide-and-conquer algorithms often require additional memory for storing intermediate results, which can be a drawback for memory-constrained environments.
3. **Complexity of Merging Results:** The merging step can be complex and may not always be straightforward. Efficient merging often requires additional algorithms and can add to the complexity of the overall solution.

4. **Not Always the Most Efficient:** For some problems, divide-and-conquer might not be the most efficient approach compared to iterative or dynamic programming methods. The choice of strategy depends on the specific problem and context.
5. **Difficulty in Implementation:** Implementing divide-and-conquer algorithms can be more challenging, especially for beginners. The recursive nature and merging steps require careful design to ensure correctness and efficiency.
6. **Stack Overflow Risk:** Deep recursion can lead to stack overflow errors if the recursion depth exceeds the system's stack capacity, particularly with large inputs or poorly designed algorithms.

 The power of divide-and-conquer lies in its efficiency; it often reduces the problem's complexity by tackling smaller parts, which can lead to significant performance improvements.

The divide-and-conquer approach is a versatile and powerful problem-solving strategy that breaks down complex problems into simpler subproblems. Its applications span various fields, including sorting, searching, and computational geometry. While it offers significant advantages in terms of efficiency and simplicity, it also comes with challenges such as recursion overhead and merge step complexity. Understanding and mastering this technique is essential for tackling a wide range of algorithmic problems.

### 13.3 Dynamic Programming Approach to Problem Solving

*“Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is a way of combining solutions to overlapping subproblems to avoid redundant calculations.”*

– Richard Bellman

Imagine you need to travel by taxi from Thiruvananthapuram to Ernakulam. You have several possible routes through different cities, and your goal is to find the one that gets you to Ernakulam in the shortest time, which you call the optimal route. Your cousin in Alappuzha knows the best route from Alappuzha to Ernakulam. Given that any optimal route from Thiruvananthapuram must pass through Alappuzha, you can simplify your task by first finding the best route from Thiruvananthapuram to Alappuzha, which is closer and has fewer possible routes. Once you have this route, you can follow the one your cousin has suggested from Alappuzha to Ernakulam.

This approach is based on the principle of optimality, which states that any optimal route from Thiruvananthapuram to Ernakulam via Alappuzha must have optimal sub-routes. Specifically, the segment from Alappuzha to Ernakulam must be the best route between these two cities, and the segment from Thiruvananthapuram to Alappuzha must also be optimal. More generally, if you have an optimal route consisting of cities  $C_1, C_2, \dots, C_p$ , then each segment of this route (from  $C_1$  to  $C_2$ ,  $C_2$  to  $C_3$ , etc.) must be optimal on its own. By solving the problem in smaller parts—finding the best route from Thiruvananthapuram to Alappuzha and using the known optimal route from Alappuzha to Ernakulam—you can effectively solve the larger problem. This principle, known as Bellman’s principle of optimality, was developed by Richard Bellman in the late 1940s and applies to various optimization problems beyond travel routes. In this section, we will explore how dynamic programming leverages this principle to tackle complex optimization challenges.

Dynamic programming (DP) is a method for solving problems by breaking them down into smaller overlapping subproblems, solving each subproblem just once, and storing their solutions. It is particularly useful for optimization problems where the problem can be divided into simpler subproblems that are solved independently and combined to form a solution to the original problem.

Dynamic Programming was first introduced by Richard Bellman in the 1950s as part of his research in operations research and control theory. In this context, the term “programming” does not relate to coding but refers to the process of optimizing a series of decisions. Bellman chose the term “dynamic programming” to avoid confusion and political issues, as “programming” was strongly associated with computers at the time.

At its core, Dynamic Programming (DP) involves breaking down a problem into smaller, more manageable subproblems and storing the solutions to these subproblems for future use. This approach is particularly effective for problems that exhibit two key properties: optimal substructure and overlapping subproblems.

1. **Optimal Substructure:** A problem has optimal substructure if the best solution to the overall problem can be constructed from the best solutions to its smaller subproblems. This means that if you have the optimal solutions for the smaller components of the problem, you can combine them to find the best solution for the entire problem. This property allows Dynamic Programming to build solutions incrementally, using previously computed results to achieve the most efficient outcome.

### Example: Shortest Path in a Grid

Imagine you need to find the shortest path from the top-left corner to the bottom-right corner of a grid. You can only move right or down. Each cell in the grid has a certain cost associated with entering it, and your goal is to minimize the total cost of the path.

**Problem Breakdown:**


- (a) **Smaller Subproblems:** To find the shortest path to a particular cell  $(i, j)$ , you can look at the shortest paths to the cells immediately above it  $(i - 1, j)$  and to the left of it  $(i, j - 1)$ . The cost to reach cell  $(i, j)$  will be the minimum of the costs to reach these neighboring cells plus the cost of the current cell.
- (b) **Optimal Substructure:** If you know the shortest paths to cells  $(i - 1, j)$  and  $(i, j - 1)$ , you can use these to determine the shortest path to cell  $(i, j)$ . The optimal path to cell  $(i, j)$  can be constructed from the optimal paths to its neighboring cells.

**How it Works:**

- You start by solving the problem for the smallest subproblems (the cells directly above and to the left).
- You then build up solutions incrementally, using the results of the smaller subproblems to solve larger parts of the grid.
- Finally, you combine the results to find the shortest path to the bottom-right corner of the grid.

This approach ensures that you are using the most efficient solutions to smaller problems to construct the best solution for the entire grid.

2. **Overlapping Subproblems:** Many problems require solving the same subproblems multiple times. Dynamic Programming improves efficiency by storing the results of these subproblems in a table to avoid redundant calculations. By caching these results, the algorithm reduces the number of computations needed, leading to significant performance improvements.

 Dynamic programming breaks problems down into overlapping subproblems, storing solutions to avoid redundant calculations.

**Example: Fibonacci Sequence**

In the Fibonacci sequence, each number is the sum of the two preceding ones. For example, to find **Fibonacci(5)**, you need the values of **Fibonacci(4)** and **Fibonacci(3)**. To compute **Fibonacci(4)**, you need **Fibonacci(3)** and **Fibonacci(2)**. Notice that **Fibonacci(3)** is computed multiple times when calculating different Fibonacci numbers.

**Without Dynamic Programming:**

- To compute **Fibonacci(5)**, you might end up calculating **Fibonacci(3)** twice.

- This redundancy leads to a lot of repeated work.


**With Dynamic Programming:**

- You compute **Fibonacci(3)** once and store its result.
- When you need **Fibonacci(3)** again, you retrieve the stored result instead of recalculating it.
- This caching of results avoids redundant calculations and speeds up the process.

**How it Works:**

- (a) **Compute:** Calculate the Fibonacci numbers and store them in an array.
- (b) **Reuse:** Whenever you need the value of a Fibonacci number that has already been computed, look it up in the array instead of recalculating.

By storing results and reusing them, Dynamic Programming reduces the number of calculations needed to solve the problem, leading to significant performance improvements.

 By caching intermediate results, dynamic programming transforms exponential time complexity into polynomial time.

### 13.3.1 Comparison with Other Problem-solving Techniques

Dynamic Programming shares similarities with other problem-solving techniques like Divide and Conquer and Greedy Algorithms, but it has unique characteristics that set it apart:

**Divide and Conquer:** Both techniques break problems into smaller subproblems. However, Divide and Conquer solves each subproblem independently, often without considering if the same subproblems are solved multiple times. In contrast, Dynamic Programming stores and reuses solutions to overlapping subproblems, which improves performance by avoiding redundant calculations.

**Greedy Algorithms:** Greedy algorithms make a series of locally optimal choices with the hope of finding the global optimum. They are typically simpler to implement but may not always yield the best overall solution. Dynamic Programming, on the other hand, guarantees an optimal solution by evaluating all possible choices and storing the best solutions for each subproblem, ensuring the most efficient overall result.



### 13.3.2 Fundamental Principles of Dynamic Programming


In this section, we will explore the fundamental principles that make Dynamic Programming an effective problem-solving technique, focusing on overlapping subproblems, optimal substructure, and the two primary approaches: memoization and tabulation.

**Overlapping Subproblems:** Dynamic Programming is particularly useful for problems with overlapping subproblems. This means that when solving a larger problem, you encounter smaller subproblems that are repeated multiple times. Instead of recomputing these subproblems each time they are encountered, Dynamic Programming saves their solutions in a data structure, such as an array or hash table. This avoids redundant calculations and significantly improves efficiency.

For example, in a recursive approach to solving a problem, the same function might be called multiple times with the same arguments. Without Dynamic Programming, this leads to wasted time as the same subproblems are recalculated repeatedly. By using Dynamic Programming, the solutions to these subproblems are stored once computed, which optimizes overall algorithm efficiency.

**Optimal Substructure:** Another key principle of Dynamic Programming is optimal substructure. This property means that an optimal solution to the larger problem can be constructed from the optimal solutions to its smaller subproblems. In other words, if you can determine the best solution for smaller problems, you can use these solutions to build the best solution for the entire problem.

*Optimal substructure* is central to Dynamic Programming's recursive nature. By solving subproblems optimally and combining their solutions, you ensure that the final solution is also optimal.

 Optimal substructure is the key to dynamic programming, where the global solution can be constructed from optimal solutions of smaller subproblems.

### 13.3.3 Approaches in Dynamic Programming

Dynamic Programming can be implemented using two main approaches: memoization (top-down) and tabulation (bottom-up).

#### 13.3.3.1 Memoization (Top-Down Approach)

Memoization involves solving the problem recursively and storing the results of subproblems in a table (usually a dictionary or array). This way, each subproblem is solved only once, and subsequent calls to the subproblem are served from the stored results.

**Steps:**

1. Identify the base cases.
2. Define the recursive relation.
3. Store the results of subproblems in a table.
4. Use the stored results to solve larger subproblems.

**Example:** Fibonacci sequence using memoization

```
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

**Code Explanation: Fibonacci Sequence Using Memoization:**

The given code defines a function **fib** that calculates the **n**-th Fibonacci number using a technique called memoization. Memoization is a method used to optimize recursive algorithms by storing the results of expensive function calls and reusing them when the same inputs occur again.

Here is a detailed explanation of each part of the code:

```
def fib(n, memo={}):
```

- The function **fib** takes two arguments:
  - **n**: The position in the Fibonacci sequence for which we want to find the Fibonacci number.
  - **memo**: A dictionary used to store previously computed Fibonacci numbers. It defaults to an empty dictionary if not provided.

```
    if n in memo:
        return memo[n]
```

- This line checks if the Fibonacci number for the given **n** has already been computed and stored in the memo dictionary.

- If it has, the function immediately returns the stored value, avoiding redundant calculations.

```
if n <= 1:
    return n
```

- This line handles the base cases of the Fibonacci sequence.
  - \* If **n** is **0** or **1**, the function returns **n** because the Fibonacci sequence is defined as **fib(0) = 0** and **fib(1) = 1**.

```
memo[n] = fib(n-1, memo) + fib(n-2, memo)
```

- This line computes the Fibonacci number for **n** by recursively calling **fib** for **n-1** and **n-2**.
- The results of these recursive calls are added together to get the Fibonacci number for **n**.
- The computed Fibonacci number is then stored in the **memo** dictionary to avoid redundant calculations in future calls.

```
return memo[n]
```

- This line returns the Fibonacci number for **n** that was just computed and stored in the **memo** dictionary.

The **fib** function leverages memoization to optimize the calculation of Fibonacci numbers by storing the results of previously computed numbers in a dictionary. This approach significantly reduces the time complexity of the algorithm from exponential to linear by avoiding redundant calculations.

Memoization is often easier to implement and understand. It starts with the original problem and solves subproblems as needed. However, it may have overhead due to recursive function calls and may not be as efficient for some problems.

### 13.3.3.2 Tabulation (Bottom-Up Approach)

Tabulation involves solving the problem iteratively and filling up a table (usually an array) in a bottom-up manner. This approach starts with the smallest subproblems and uses their solutions to construct solutions to larger subproblems.

**Steps:**

1. Identify the base cases.
2. Define the table to store solutions to subproblems.
3. Fill the table iteratively using the recursive relation.
4. Extract the solution to the original problem from the table.

**Example:** Fibonacci sequence using tabulation

```
def fib(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

**Code Explanation: Fibonacci Sequence Using Table:**

The given code defines a function **fib** that calculates the **n**-th Fibonacci number using a tabular approach. Dynamic programming is a method for solving problems by breaking them down into simpler subproblems and storing the solutions to these subproblems in a table to avoid redundant calculations.

Here is a detailed explanation of each part of the code:

```
def fib(n):
```

- The function **fib** takes a single argument **n**, which represents the position in the Fibonacci sequence for which we want to find the Fibonacci number.

```
    if n <= 1:
        return n
```

- This line handles the base cases of the Fibonacci sequence.
  - \* If **n** is **0** or **1**, the function returns **n** because the Fibonacci sequence is defined as **fib(0) = 0** and **fib(1) = 1**.

```
    dp = [0] * (n + 1)
```