MODULE 4

COMPUTATIONAL APPROACHES TO PROBLEM-SOLVING

COMPUTATIONAL APPROACHES TO PROBLEM-SOLVING

- Computational problem-solving is a systematic approach to formulating, analyzing, and solving problems through algorithms, logic, and computational power.
- It is a cornerstone of computer science and engineering, offering structured methods to handle challenges in a variety of fields.
- Computational approaches to problem-solving involve using structured methods and algorithms to tackle problems efficiently.

WHY DIFFERENT APPROACHES ?

- The diversity of computational approaches allows us to adapt problem-solving strategies to the specific nature of the problem, constraints, and desired outcomes.
- Using the right approach not only ensures efficiency but also balances resource utilization and practical feasibility.
- This flexibility is crucial in both theoretical and real-world applications.

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

BRUTE-FORCE APPROACH

- The **brute force approach** is one of the most basic and intuitive methods for solving computational problems.
- It involves systematically trying all possible solutions or combinations to find the correct or optimal one.
- This approach is guaranteed to work for any problem where a solution exists, but it may not be practical for larger or more complex problems due to its inefficiency.
- 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.

BRUTE-FORCE APPROACH

- 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.

EXAMPLES OF BRUTE-FORCE APPROACH

Padlock

- If you come across a padlock with a four-digit numeric code, the brute-force approach would require trying every possible combination from "0000" to "9999" in sequence until the correct code is found.
- While this method is straightforward and ensures the correct combination is eventually discovered, it can be slow.

EXAMPLES OF BRUTE-FORCE APPROACH

Password Guessing

- In cybersecurity, brute-force attacks are employed to break passwords by methodically trying every possible combination of characters until the correct one is found.
- This method works well against weak passwords that are short or lack complexity.
- For example, attacking a six-character password made up of letters and digits would require testing all 2.18 billion (36^6) potential combinations to find the right one.

EXAMPLES OF BRUTE-FORCE APPROACH

Cryptography: Cracking Codes

- In cryptography, brute-force attacks are used to break codes or encryption keys by exhaustively testing every possible combination until the correct one is discovered.
- For instance, decrypting a simple substitution cipher involves trying all possible shifts in the alphabet until the plaintext message is revealed.

EXAMPLES OF BRUTE-FORCE APPROACH

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.

CHARACTERISTICS OF BRUTE-FORCE SOLUTIONS

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

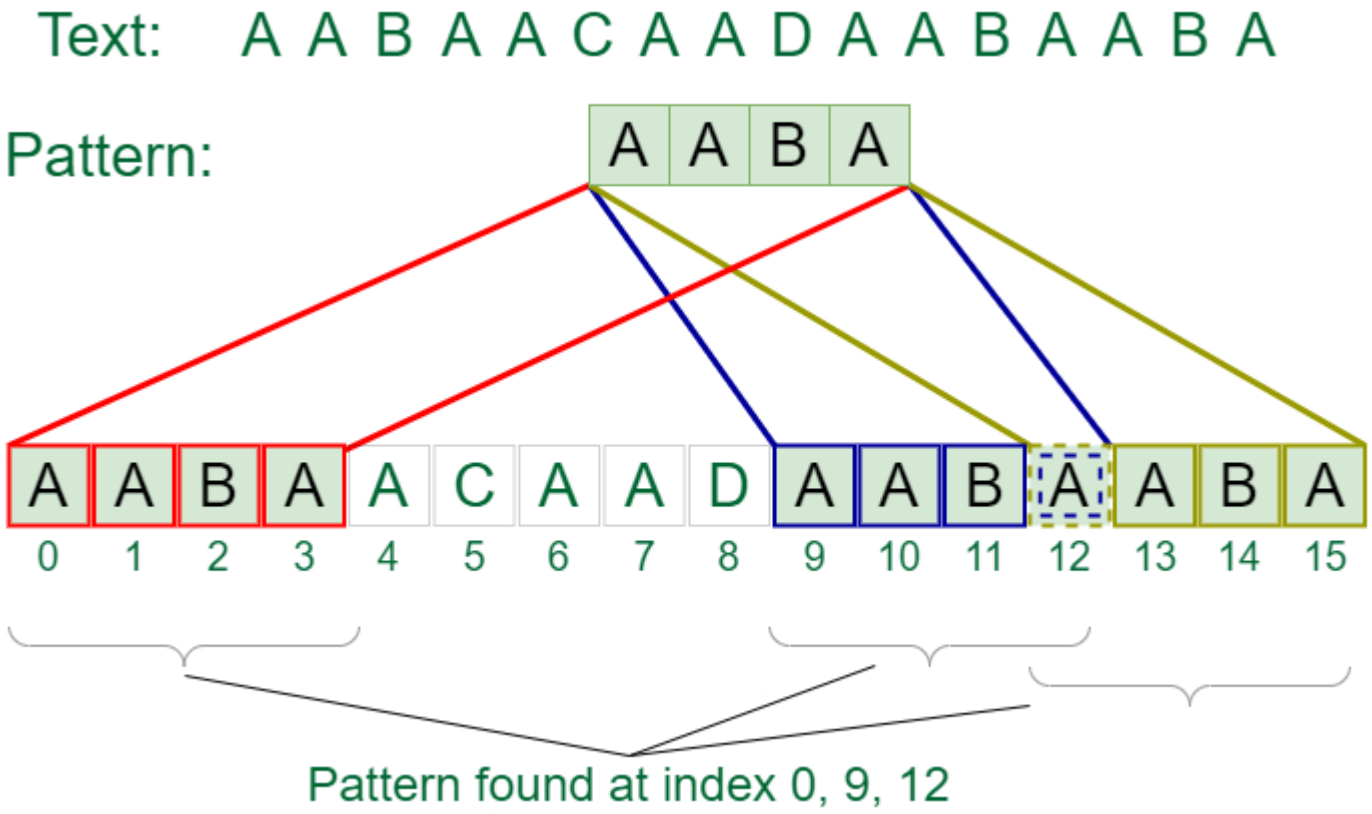
SOLVING COMPUTATIONAL PROBLEMS USING BRUTE-FORCE APPROACH

To solve computational problems using the brute-force approach:

- **Define the problem clearly**: Understand the problem and its requirements.
- **Generate all possible solutions**: Consider every potential candidate for the solution.
- **Evaluate each solution**: Check each candidate against the problem's criteria
- **Identify the correct solution**: Once a valid solution is found, it is selected as the answer.

SOLVING COMPUTATIONAL PROBLEMS USING BRUTE-FORCE APPROACH APPROACH

PROBLEM-I (STRING MATCHING)



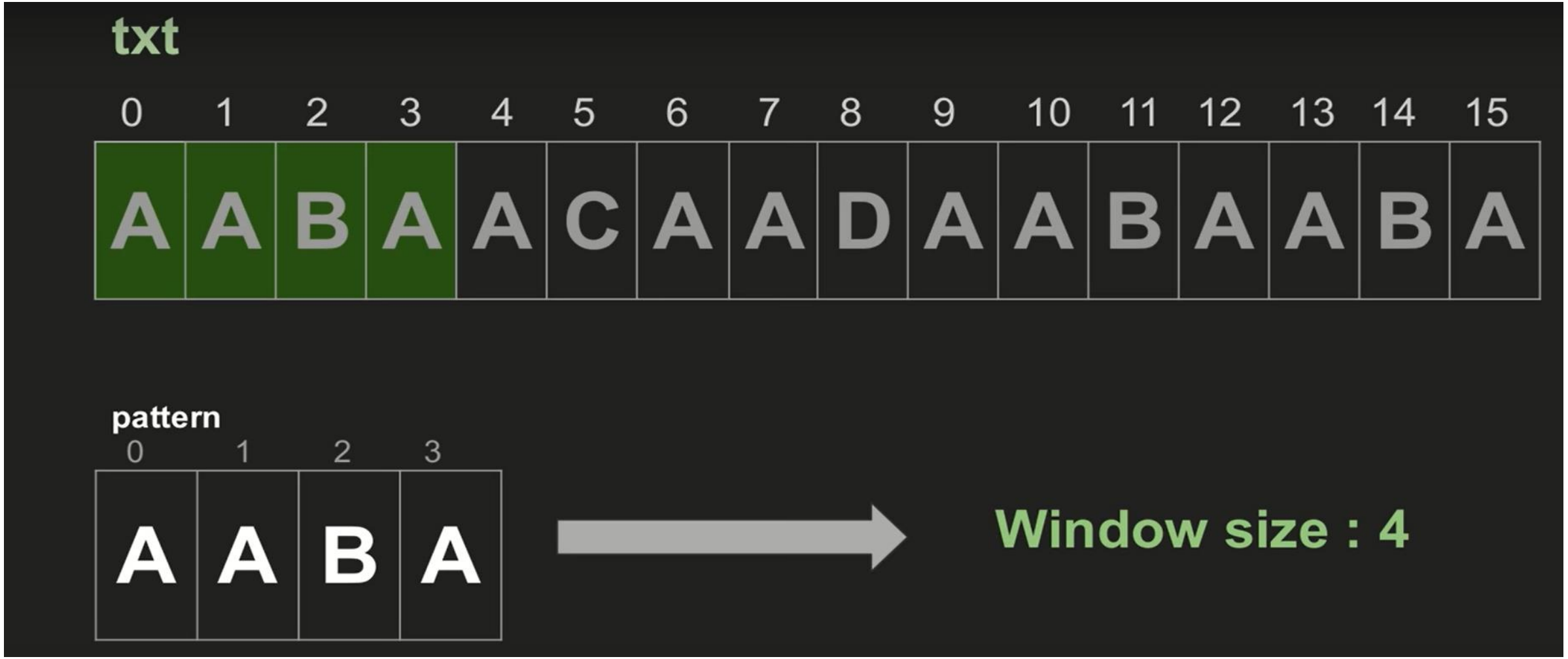
PROBLEM-1 (STRING MATCHING)

- The brute-force string matching algorithm is a simple method for finding all occurrences of a pattern within a text.

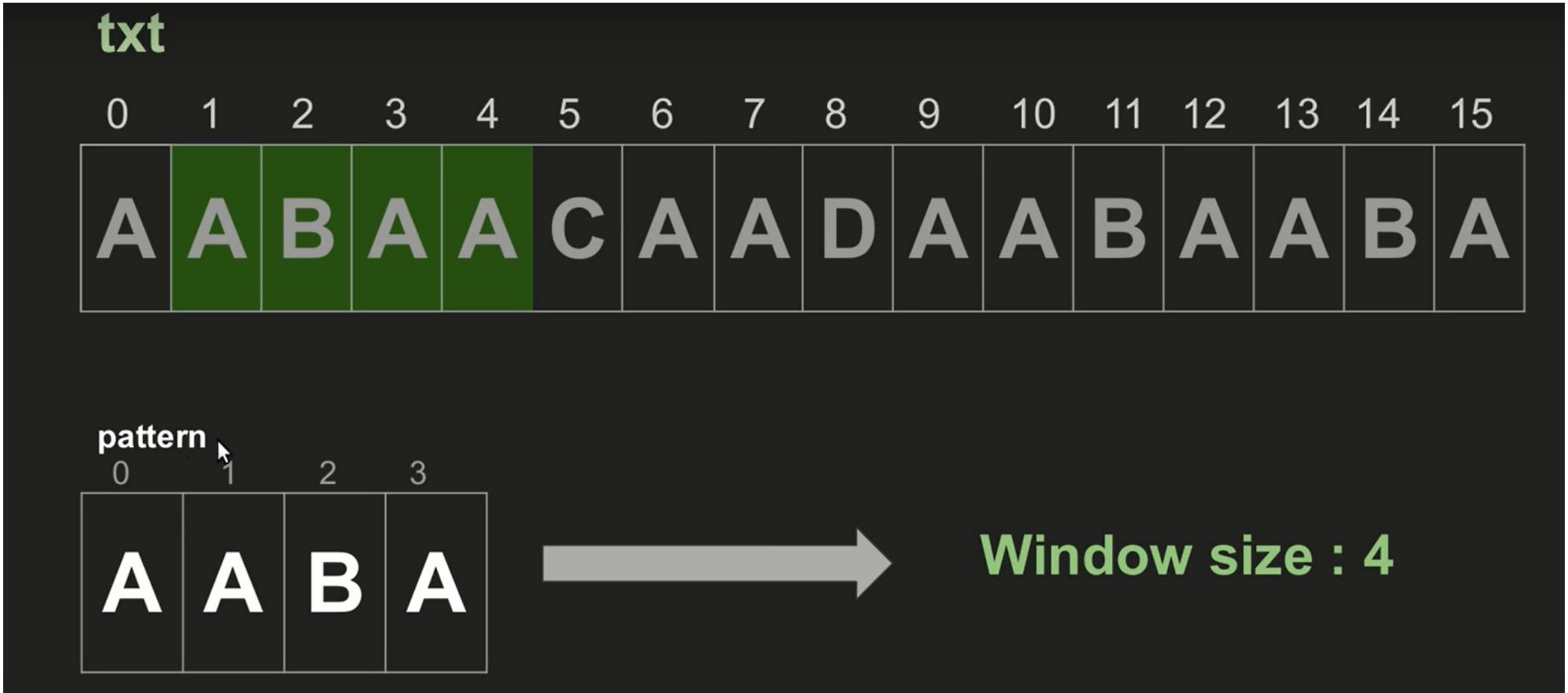
How it works?

- **Start at the beginning of the text**: Align the pattern with the first character of the text.
- **Check for a match**: Compare the pattern with the substring of the text starting at the current position. If it matches, record the position.
- **Move to the next position**: Shift the pattern one character to the right and repeat the comparison.
- **Finish**: Continue checking all positions in the text until the end is reached.

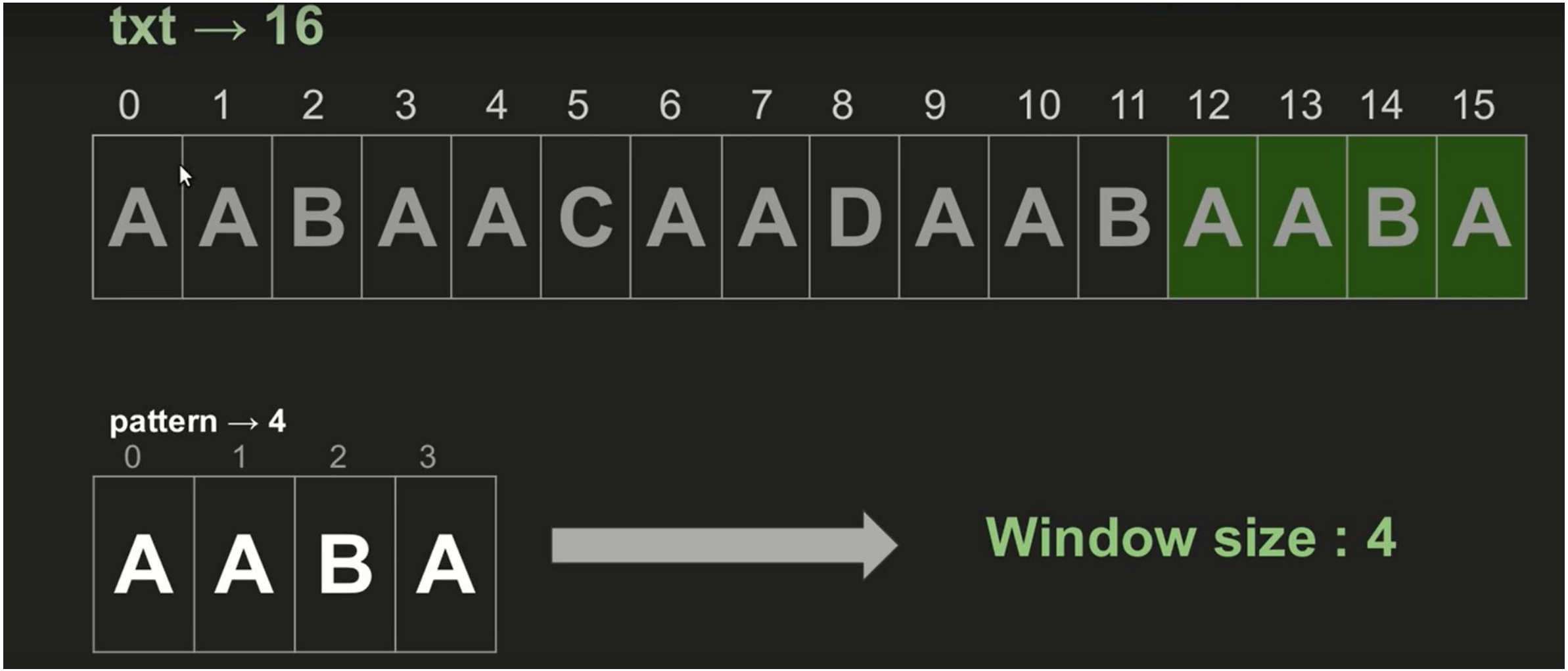
PROBLEM-1 (STRING MATCHING)



PROBLEM-1 (STRING MATCHING)



PROBLEM-I (STRING MATCHING)



PYTHON - IMPLEMENTATION

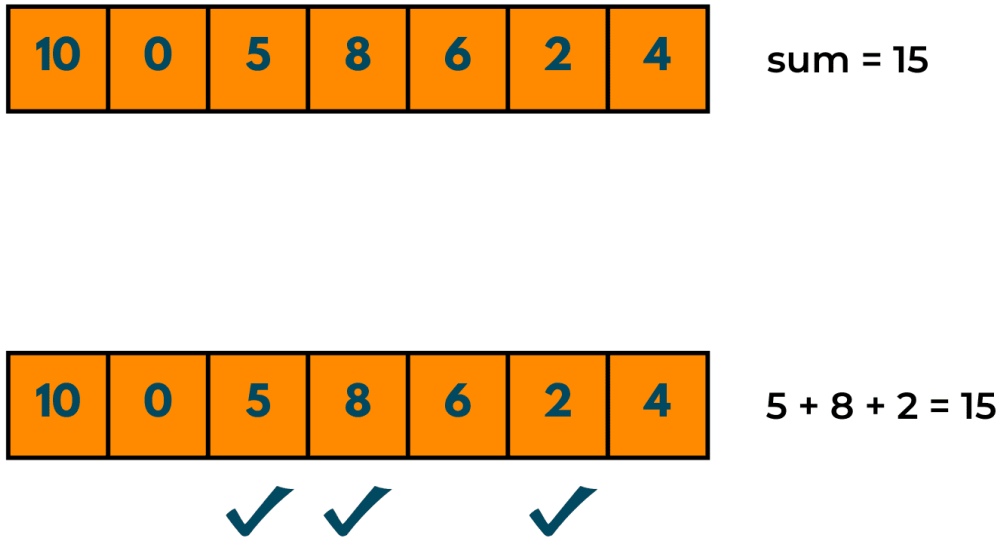
```
def brute_force_string_match(text, pattern):
n = len(text)  # Length of the text
m = len(pattern)  # Length of the pattern
# Loop over each possible starting index in the text
for i in range(n - m + 1):
        substring = text[I : i + m]  # Extract the substring of the text from the current position
        # 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)
```

OUTPUT:

```
Pattern found at index 10
```

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.



PROBLEM-2 (SUBSET SUM PROBLEM)

How the brute-force approach works?

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

PROBLEM-2 (SUBSET SUM PROBLEM)

```
def subset_sum_brute_force(nums, target):
    n = len(nums)
    # Loop over all possible subsets
    for i in range(1 << n):
        subset = [nums[j] for j in range(n) if (i & (1 << j))] # Extract subset using the binary representation of I
        if sum(subset) == target:
            return subset
    return None # Return None if no subset with the target sum is found

# 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

- nums = [3, 34, 4, 12, 5, 2]
- target = 9
- The goal is to find a subset of nums that sums up to 9
- The number of subsets is 2^n where n is the length of the list.
- So, there are 2^6=64 subsets in total.
- The subsets are generated by using all integers from 0 to 63 ((binary numbers from 000000 to 111111)
- Where each bit represents whether the corresponding element in nums is included in the subset
- We will iterate through all these 64 numbers and generate subsets by checking each bit of the number.

Iteration Details:

- **i = 0 (binary 000000)**:
  - No elements selected.
  - Subset:[], Sum=0, Not equal to 9
- **i = 1 (binary 000001)**:
  - Select element nums[0] = 3
  - Subset: [3], Sum: 3, Not equal to 9

….

PROBLEM-3 (SUDOKU SOLVER)

- The brute-force approach to solving a Sudoku puzzle involves trying every possible number in each empty cell until the puzzle is solved.
- **Find an empty cell**: Look for the first empty cell in the puzzle.
- **Try all numbers (1–9)**: Place each number from 1 to 9 in the empty cell and check if it follows the Sudoku rules.
- **Check for validity**: The number is valid if it does not repeat in the same row, column, or 3x3 subgrid.
- **Move to the next empty cell**: If the number is valid, move to the next empty cell and repeat.
- **Backtrack if needed**: If placing a number leads to a conflict later, undo the previous step and try the next number.

SUDOKU SOLVER – PYTHON CODE

https://colab.research.google.com/drive/1yjTaiFu998kaPWdfPLbRMq0FKKFyM1Fi?usp=sharing

ADVANTAGES AND LIMITATIONS OF BRUTE-FORCE APPROACH

- Guaranteed Solution - methods ensure finding a solution if one exists
- Simplicity: - Straightforward to implement and understand, requiring minimal algorithmic complexity.
- Versatility: - Applicable across various domains where an exhaustive search is feasible

- Computational Intensity - It can be highly resource-intensive, especially for problems with large solution spaces.
- Time Complexity - long execution times to find solutions.
- Scalability Issues - In scenarios with exponentially growing solution spaces, brute-force methods may become impractical or infeasible to exe cute within reasonable time constraints

OPTIMIZING BRUTE-FORCE SOLUTIONS

- Pruning: Eliminate certain candidates early if they cannot possibly be a solution.
- 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.
- Dynamic Programming: Store the results of subproblems to avoid re dundant computations.

THANK YOU