CESI_
ÉCOLE D'INGÉNIEURS

CER 5

Balance of production and consumption

Part 4

Date: 26/09/2024

Prosit : Balance of production and consumption

Prosit Info:

- ***Generalisation :***
    - *temporal & spatial complexity.*
    - *Data structures (hash map).*
    - *Algorithms*
- ***Context :*** *A company has tasked us to produce a more optimised algorithm for a smart grid.*
- **Animateur :** Clement
- **Scribe :** Junior
- **Secretary :** Palal
- **Manager :** Peio

Keywords

- **spatial complexity** : refers to the amount of memory or space a program or algorithm requires to execute, relative to the size of the input.
- **brute force** : Brute force in programming refers to a problem-solving technique where you try every viable option to find a solution
- **hash table / hash maps** is a data structure that uses hashing to map keys to values. The hash code generated from the hash function determines where the corresponding value is stored in memory. Hash tables allow for efficient data lookup, insertion, and deletion.
- **algorithm** : is a finite set of well-defined, step-by-step instructions or rules used to solve a problem or perform a task.
- **time / (temporal) complexity** is a measure of the amount of time an algorithm takes to complete as a function of the size of its input.

("Understanding Time Complexity: A Beginner's Guide") It is used to evaluate the performance and efficiency of an algorithm, particularly how its running time grows as the input size increases.

Constraints

- faster algorithm time (> o(n^2))
- large volume of data
- temporal & spatial complexity

Issue

How can we produce a search algorithm faster than the brute force approach?

Hypothesis

- o(n log(n))
- O(n) complexity is not possible
- as temporal complexity reduces, spatial complexity increase
- use of iteration (loops)
- use of hash maps & hash table
- use of recursion
- cannot use n!

Deliverables

- python script for the algorithm

Action plan

1. Keywords
2. WS
3. Resource Analysis
4. Brute force
5. Temporal, time & spatial complexity
6. analyse a sample of the dataset
7. (sorting of the data)
8. create an algorithm
9. optimisation of the algorithm
10. (optimise the brute force)
11. prove time & spatial complexity
12. comparison of algorithms so as to produce a lower complexity type
13. comparison with brute force
14. Validation of assumptions
15. Conclusion

Resource Analysis

Below is a table with several types of algorithms with their corresponding complexity (spatial & temporal) ranging from best to work case.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Quick Sort	O(n log(n))	O(n log(n))	$O(n^2)$	O(n)
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Bucket Sort	O(n +k)	O(n +k)	$O(n^2)$	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Count Sort	O(n +k)	O(n +k)	O(n +k)	O(k)
Shell Sort	O(n log(n))	O(n log(n))	$O(n^2)$	O(1)
Tim Sort	O(n)	O(n log(n))	O(n log (n))	O(n)
Tree Sort	O(n log(n))	O(n log(n))	$O(n^2)$	O(n)
Cube Sort	O(n)	O(n log(n))	O(n log(n))	O(n)

Measuring Space Complexity

The space complexity of an algorithm is expressed in terms of Big O notation, which describes the upper bound of space usage in terms of the input size n. It helps to understand how much memory an algorithm might need for various input sizes.

- O(1): Constant space. The algorithm requires a fixed amount of memory regardless of input size. (“Why is Analysis of Algorithms important? — Klu”)
- O(n): Linear space. The memory usage grows linearly with the size of the input. (“Linear Space Complexity - Reintech”)
- O(n^2): Quadratic space. The space required grows proportionally to the square of the input size.

Key Concepts of Time Complexity

1. **Input Size (n):**
    - Time complexity is typically expressed as a function of the size of the input, denoted as n. The size could represent the number of elements in a list, the number of nodes in a tree, or the number of characters in a string, depending on the problem.
2. **Operations Count:**
    - Time complexity is based on the number of basic operations (like comparisons, assignments, arithmetic operations, etc.) that the algorithm performs. Instead of exact time (in seconds or milliseconds), the focus is on the *rate of growth* of operations as the input size increases.
3. **Big O Notation:**
    - Time complexity is expressed using Big O notation, which describes the upper bound of an algorithm's running time in terms of input size. (“What is Time Complexity And Why Is It Essential?”) It helps compare the efficiency of different algorithms, especially for large inputs.
    - Some common time complexities in Big O notation are:

- **O(1): Constant time —** the running time does not change with the size of the input.
- **O(log n): Logarithmic time —** the running time grows logarithmically with the input size. **("Mastering Algorithm Complexity: Time & Space Optimization")**
- **O(n): Linear time** — the running time grows linearly with the input size**. ("Mastering Algorithm Complexity: Time & Space Optimization")**
- **O(n log n): Linearithmic time —** the running time grows at a rate of n * log(n).
- **O(n^2): Quadratic time —** the running time grows quadratically with the input size.
- **O(2^n): Exponential time** — the running time doubles with each additional input element**. ("Exploring Algorithms for Arrays and Strings: Two Pointers ... - Medium")**
- **O(n!): Factorial time —** the running time grows factorially with the input size. **(" Computational Complexity Theory Unit 3 - Fiveable")**

Common Applications of Brute Force

1. **Password Cracking**: Trying every possible password combination until the correct one is found. ("Brute Force Attack vs. Dictionary Attack - This vs. That")
2. **Searching Problems**: Trying every possible arrangement or permutation of elements.
3. **String Matching**: Comparing a substring with each possible substring in the target text.

Performance of Brute Force

Brute force solutions typically have poor time complexity:

- **Password cracking**: Time complexity is often O(k^n), where k is the size of the character set and n is the length of the password.
- **String matching**: O(n * m), where n is the length of the text and m is the length of the pattern. (“Naive algorithm for Pattern Searching - GeeksforGeeks”)
- **Subset generation**: O(2^n) since there are 2^n subsets.
- **Permutation generation**: O(n!) since there are n! permutations of a list of size n.

When to Use Brute Force

- When the problem size is small.
- When an exact solution is required, and other algorithms may be too complex.
- When a more efficient algorithm is difficult to implement.

Types of Algorithms

1. **Greedy Algorithms**: These make the locally optimal choice at each step with the hope of finding a global optimum. (“What is brute-force search? - Klu”) Example: Dijkstra's algorithm for shortest paths.
2. **Divide and Conquer**: This involves breaking a problem into smaller sub-problems, solving each one individually, and combining the solutions. Example: Merge sort and Quick sort.
3. **Dynamic Programming**: It breaks problems down into smaller overlapping sub-problems and stores solutions to avoid redundant work. Example: Fibonacci sequence and Knapsack problem.
4. **Backtracking**: A brute-force search method that builds a solution incrementally, abandoning solutions that fail to satisfy the problem constraints. Example: Solving mazes, N-Queens problem.
5. **Recursive Algorithms**: These call themselves with modified inputs to solve a smaller instance of the same problem. Example: Factorial calculation, Binary search.

6. **Sorting Algorithms**: Algorithms designed to organize data in a specific order. Examples include Bubble sort, Merge sort, and Quick sort.

Temporal & spatial complexity

The relationship between time complexity and space complexity isn't always as simple as saying "as temporal complexity reduces, spatial complexity increases." However, there are **trade-offs** between time and space complexity in many algorithms, though it's not a strict rule.

Key Points:

1. **Time Complexity** refers to how the running time of an algorithm increases as the size of the input increases.
2. **Space Complexity** refers to how much memory an algorithm uses as the size of the input increases.

Common Scenarios of Time-Space Trade-off:

In many cases, you can reduce time complexity at the expense of increasing space complexity (and vice versa). This is known as the **time-space trade-off**. However, it is not universally true that improving one always worsens the other; it depends on the specific algorithm and its approach.

Example 1: Caching / Memorisation (Dynamic Programming)

- **Memorisation** stores previously computed results to avoid redundant calculations, improving **time complexity** at the cost of using additional memory.
    - **Time Complexity**: Improved by caching results, avoiding precomputations.
    - **Space Complexity**: Increased by storing intermediate results.

For example, recursive algorithms like Fibonacci can be optimized using memorization:

- Without memorization: Time Complexity is **O(2^n),** and Space Complexity is **O(n)** (due to recursion stack).
- With memorization: Time Complexity becomes **O(n)**, but Space Complexity increases to **O(n)** because we store results of previous computations.

Example 2: Pre-computation

- **Pre-computation** of results (e.g., using hash tables or look-up tables) improves runtime by allowing constant-time lookups, but it requires extra memory to store precomputed data.
  - **Time Complexity**: Decreases because lookups are faster (e.g., **O(1)** for hash tables).
  - **Space Complexity**: Increases because the data needs to be stored.

Example 3: Brute Force vs Optimized Algorithms

- A brute force algorithm might have:
  - **Time Complexity**: $O(n^2)$
  - **Space Complexity**: O(1) (because no extra memory is needed).

But an optimized algorithm using additional data structures (e.g., hash maps) might:

- **Time Complexity**: O(n)
- **Space Complexity**: O(n) (due to the use of the hash map).

Example 4: Sorting Algorithms

- **Merge Sort** and **Quick Sort**:
  - Merge Sort has a time complexity of **O(n log n)**, but it requires additional memory for temporary arrays, leading to space complexity of **O(n)**.
  - Quick Sort also has time complexity **O(n log n)**, but its space complexity can be as low as **O(log n)** (without additional storage).

Summary:

- **Time-space trade-off** is a common concept, where reducing time complexity often leads to higher space complexity and vice versa, but it is not a universal law. Some algorithms improve both time and space complexities simultaneously, and others don't exhibit any trade-off.
- The relationship between time and space complexities depends on the specific algorithm, the problem being solved, and the approach taken. Optimizing an algorithm often involves carefully balancing time and space based on the context and constraints of the problem.

Solution

My algorithm is in soluce.py, this hereby proves the temporal & spatial complexity of my algorithm:

Step 1: Sorting the datasets in a excel file

Step 2: Loading all CSV files and gather all the numbers into a single list

Step 3: Process each number and look for pairs that form the target sum

Step 4: Calculate the complement needed to reach the target

Step 5: Check if the complement has already been seen

Step 6: The store the pair

Step 7: Check if the complement has already been seen

Step 8 : Prompt user for target input

Step 9 : Print the pairs that sum to the target

Comparison with Brute Force:

Brute Force Approach:

A brute-force solution for finding pairs that sum to a target would involve:

1. **Iterating through each number** and checking every other number to see if they sum to the target.

2. **Time Complexity**: O(n^2), where **n** is the total number of integers. This arises because for each element, you need to check all other elements, leading to nested loops.
3. **Space Complexity**: O(1) since it does not require any extra space beyond storing the input data and result pairs.

Optimized Algorithm (Using Hashing):

The optimized approach improves upon brute force by:

1. Using a **hash map** (or set) to store previously seen values. For each value, it computes the complement needed to reach the target, checks if that complement has been seen before, and stores the result if a pair is found.
2. **Time Complexity**: O(n), where **n** is the total number of integers across all CSV files. This is because we only need to iterate through the integers once, and each lookup or insertion into the hash map takes constant time, O(1).
3. **Space Complexity**: O(n) because the algorithm needs to store up to **n** values in the hash map.

Temporal Complexity (Time Complexity)

Step 1: Reading all CSV files (Line 7: data = pd.read_csv(csv_file)):

- We need to open and read each CSV file. Assuming there are k CSV files and the total number of integers in all CSV files is n.
    - **Cost of reading each CSV file**: O(file size).
    - **Total cost of reading all CSV files**: O(n), where **n** is the total number of integers across all files. This step reads the file into memory (as a Data Frame) and loads all integers from the values column into a list.

Step 2: Iterating through each number in the CSV files (Line 8: for val in num_list)

- After reading the integers into a list (num_list), we iterate over each integer.

- **Total cost of iterating through all integers**: O(n), where **n** is the total number of integers across all CSV files. There is exactly one loop iteration per integer.

Step 3: Hashing Operations (Lines 10-14):

- In each iteration, two operations happen:
  1. **Checking if complement is in seen_values (Line 11)**: Checking membership in a set is an O(1) operation on average because sets in Python are implemented as hash tables.
  2. **Inserting val into seen_values (Line 14)**: Inserting into a set is also an O(1) operation on average.

Thus, for each integer, we perform two constant-time operations:

- **Total cost for all hashing operations**: O(n) (since there are n integers).

Total Time Complexity:

- Reading CSV files: O(n)
- Iterating through all integers: O(n)
- Hashing operations: O(n)

Combining these, the total time complexity is:

O(n)

Where **n** is the total number of integers in all CSV files.

Spatial Complexity (Space Complexity)

Step 1: Memory for Reading CSV Files (Line 7: data = pd.read_csv(csv_file)):

- The Data Frame (data) will store all integers from the CSV files. If we have n total integers in all CSV files, the Data Frame will take up **O(n)** space.

Step 2: Storing Integers in Lists (Line 8: num_list = data['values'].tolist()):

- After reading, we convert the values to a list. This list holds n integers, so its space complexity is also **O(n)**.

Step 3: Hash Set for Storing Seen Values (Line 9: seen_values = set()):

- We maintain a set called seen_values to store all unique integers encountered so far. In the worst case, the set will hold all n integers. The space complexity of storing n elements in a set is **O(n)**.

Step 4: Storing Matching Pairs (Line 12: pairs_found.append((val, complement))):

- The list pairs_found stores the pairs that sum to the target. In the worst case, all numbers could form pairs (though this is rare). In the worst case, we could store **n/2** pairs, each of which consists of 2 numbers, so the space complexity would be **O(n)**.

Total Space Complexity:

- Storing the DataFrame and list: O(n)
- Storing the set (seen_values): O(n)
- Storing the result list (pairs_found): O(n)

Thus, the total space complexity is:

$$\mathbf{O(n)}$$

Where **n** is the total number of integers across all CSV files.

Conclusion

The optimized algorithm provides significant improvements over the brute-force method. By utilizing hash maps, we were able to reduce the time complexity to **O(n)** while processing the given data. This makes the algorithm scalable and suitable for large datasets, where brute-force would be inefficient and slow.