CS 2500: Algorithms

Lecture 9: Program Correctness and Sorting: Part I

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

September 17, 2024

Quick assignment: Problem 1: Recursive algorithm for mode

Problem Statement. You are tasked with writing a recursive algorithm that computes the mode of a given list of integers. The mode is the element that appears most frequently in the list. If there are multiple modes, the algorithm should return one of them.

1. **Recursive Algorithm.** Implement a recursive algorithm in Python that finds the mode of the list.
2. **Recurrence Relation.** Analyze the time complexity of your recursive algorithm by deriving the recurrence relation.
3. **Proof of correctness.** Prove the correctness of your algorithm using mathematical induction.

Quick assignment: Problem 1: Recursive algorithm for mode

Recursive Algorithm

```
Function ModeRecursive(lst, freq_dict=None, idx=0):
    if freq_dict == None then
        freq_dict ← {}
    if idx == len(lst) then
        max_freq ← -1
        mode_element ← None
        foreach (element, frequency) in freq_dict do
            if frequency > max_freq then
                max_freq ← frequency
                mode_element ← element
        return mode_element
    current_element ← lst[idx]
    if current_element in freq_dict then
        freq_dict[current_element] ← freq_dict[current_element] + 1
    else
        freq_dict[current_element] ← 1
    return ModeRecursive(lst, freq_dict, idx+1)
```

Quick assignment: Problem 1: Recursive algorithm for mode

Base Case

- Consider the case where the input list `lst` is empty, i.e., `len(lst)` $= 0$.
- In this case, the recursive function reaches the base case immediately because the index $idx = 0$ is equal to the length of the list.
- At this point, the algorithm initializes a variable `max_freq = -1` and `mode_element = None`, and returns `None`, as there are no elements to process.
- This is the expected behavior because the mode of an empty list is undefined (no elements exist). Thus, the base case is correct.

Quick assignment: Problem 1: Recursive algorithm for mode

Inductive Hypothesis

- Suppose the algorithm works correctly for a list of size n, meaning it correctly computes the mode of any list with n elements.
- This means that for a list of size n, after processing all elements, the frequency dictionary `freq_dict` stores the correct counts for all elements, and the algorithm correctly identifies the element with the highest frequency as the mode.

Quick assignment: Problem 1: Recursive algorithm for mode

Inductive Step: List of Size $n + 1$

- The recursive function starts by processing the $(n+1)^{\text{th}}$ element of the list.
- It checks if this element exists in the frequency dictionary:
  - If the element is already in the dictionary, it increments its frequency.
  - If the element is not in the dictionary, it adds the element with an initial count of 1.
- After processing the $(n+1)^{\text{th}}$ element, the algorithm makes a recursive call to process the remaining n elements (sublist up to index n), which, by the inductive hypothesis, is assumed to work correctly.
- Once the recursion returns, the algorithm compares the frequencies in the dictionary and identifies the element with the highest count as the mode.

Quick assignment: Problem 1: Recursive algorithm for mode

Conclusion of Inductive Step

- The algorithm correctly processes the $(n+1)^{\text{th}}$ element and updates the frequency dictionary.
- By the inductive hypothesis, the algorithm works correctly for the remaining n elements.
- Therefore, the entire list of size $n+1$ is processed correctly, and the algorithm identifies the mode accurately.
- By the principle of mathematical induction, the recursive algorithm correctly computes the mode for any list of size $n \geq 0$.

Quick assignment: Problem 1: Recursive algorithm for mode

Breakdown of the Recursive Algorithm

- **Base Case**: When the list is fully processed (i.e., $idx = len(lst)$).
  - In this case, we compute the mode by iterating over the frequency dictionary.
  - This takes $O(U)$ time, where U is the number of unique elements in the list.
- **Recursive Case**:
  - For each element in the list, we either update the frequency dictionary or add the element with a count of 1.
  - This step takes $O(1)$ time for each element (assuming average-case performance of a hash table).
  - The recursion then processes the next element.

Quick assignment: Problem 1: Recursive algorithm for mode

Recurrence Equation

- Let $T(n)$ represent the time complexity of the algorithm for a list of size n.
- The recurrence relation for this algorithm can be expressed as:

$$T(n) = T(n-1) + O(1) \quad \text{for} \quad n > 0$$

- The base case, when $n = 0$, takes $O(U)$ time to compute the mode:

$$T(0) = O(U)$$

- Therefore, the full recurrence equation is:

$$T(n) = \begin{cases} O(U) & \text{if } n = 0 \\ T(n-1) + O(1) & \text{if } n > 0 \end{cases}$$

Quick assignment: Problem 1: Recursive algorithm for mode

Solving the Recurrence. Let's solve the recurrence equation step by step:

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ T(n-1) &= T(n-2) + O(1) \\ T(n-2) &= T(n-3) + O(1) \\ &\vdots \\ T(1) &= T(0) + O(1) \end{aligned}$$

Substituting these values back into the original equation:

$$T(n) = T(0) + O(1) + O(1) + \cdots + O(1) \quad \text{(n terms)}$$

Since $T(0) = O(U)$, we get:

$$T(n) = O(U) + O(n)$$

Quick assignment: Problem 1: Recursive algorithm for mode

Final Time Complexity

- The final time complexity of the recursive algorithm is $O(n+U)$, where:
  - n is the total number of elements in the list.
  - U is the number of unique elements in the list.
- In the worst case (when all elements are unique, $U=n$):

$$T(n) = O(n)$$

- If there are few unique elements ($U \ll n$), the time complexity is still dominated by $O(n)$.

Problem 2: Sum of Digits

Problem Statement: Sum of Digits

- Given a number n, we want to find the sum of its digits recursively.
- Example: $n = 123$, the sum of digits is $1 + 2 + 3 = 6$.
- We aim to define a recursive algorithm, derive its recurrence equation, solve the equation, and prove its correctness.

Problem 2: Sum of Digits

Recursive Algorithm: Sum of Digits

- The recursive idea is to:
  - Extract the last digit of n (i.e., $n\%10$).
  - Recursively compute the sum of the digits of the quotient $n//10$.
  - Add these two results to get the sum.

Algorithm SumOfDigits(n)

```
if n = 0 then
    return 0                          ▷ Base case: no digits left.
end if
return (n mod 10) + SumOfDigits(n//10)
```

Problem 2: Sum of Digits

Breakdown of the Recursive Algorithm

- **Base Case**: When $n = 0$, the sum of digits is 0, so the function returns 0.
- **Recursive Case**:
  - The function extracts the last digit using $n\%10$ (constant time $O(1)$).
  - The function then recursively processes the quotient $n//10$, reducing n by one digit.

Problem 2: Sum of Digits

Recurrence Equation

- Let $T(n)$ represent the time complexity of the algorithm for a number n with d digits.
- Each recursive step processes one digit and calls the function for the number $n//10$ (i.e., removing the last digit).
- The recurrence relation is:

$$T(d) = T(d-1) + O(1)$$

- The base case occurs when $d = 0$:

$$T(0) = O(1)$$

Problem 2: Sum of Digits

Solving the Recurrence

$$
\begin{aligned}
T(d) &= T(d-1) + O(1) \\
T(d-1) &= T(d-2) + O(1) \\
T(d-2) &= T(d-3) + O(1) \\
&\vdots \\
T(1) &= T(0) + O(1)
\end{aligned}
$$

Substituting these values back into the original equation:

$$T(d) = T(0) + O(1) + O(1) + \cdots + O(1) \quad \text{(d terms)}$$

Since $T(0) = O(1)$, we get:

$$T(d) = O(1) + O(d) = O(d)$$

Problem 2: Sum of Digits

Proof of Correctness: Mathematical Induction. We will prove the correctness of the algorithm using **mathematical induction**.

Base Case:

- When $n = 0$, the algorithm returns 0.
- This is correct, as the sum of the digits of 0 is indeed 0.

Problem 2: Sum of Digits

Inductive Step

- **Inductive Hypothesis**: Assume that the algorithm correctly computes the sum of digits for any number with d digits.
- **Inductive Step**:
  - For a number with $d+1$ digits, the algorithm:
    - Extracts the last digit using $n\%10$.
    - Recursively computes the sum of the first d digits using $n//10$.
- By the inductive hypothesis, the algorithm correctly computes the sum of the first d digits.

Conclusion of Inductive Step

- After computing the sum of the first d digits, the algorithm adds the last digit (which is correctly computed as $n\%10$).
- Thus, the algorithm correctly computes the sum of all $d+1$ digits.

Therefore, by **mathematical induction**, the algorithm is correct for any number n.

Problem 3: Swap Every Two Adjacent Nodes

Problem Statement

- Given a linked list, swap every two adjacent nodes and return the new head of the list.
- You must solve the problem without modifying the values of the nodes; only the pointers (nodes themselves) can be changed.

Problem 3: Swap Every Two Adjacent Nodes

Linked List Before Swapping

$\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3} \rightarrow \boxed{4}$

Linked List After Swapping Adjacent Nodes

$\boxed{2} \rightarrow \boxed{1} \rightarrow \boxed{4} \rightarrow \boxed{3}$

Recursive Algorithm - Intuition

- Recursively swap the first two nodes and continue swapping the rest.
- Base case: If there are fewer than two nodes, no swap is needed.
- Recursive case: Swap the first two nodes, and recursively call the function to handle the remaining nodes.

Problem 3: Swap Every Two Adjacent Nodes

Algorithm SwapPairs(head)

```
if head = None or head.next = None then
    return head                          ▷ Base case: fewer than two nodes
end if
first ← head
second ← head.next
first.next ← SwapPairs(second.next)      ▷ Recursive call
second.next ← first                      ▷ Swap the two nodes
return second
```

Problem 3: Swap Every Two Adjacent Nodes

Breakdown of Recursive Algorithm

- **Base Case:** If there are fewer than two nodes left (i.e., no node or only one node), return the head as no swaps are needed.
- **Recursive Case**:
  - Assign the first node to `first` and the second node to `second`.
  - Call the function recursively on the next pair starting from `second.next`.
  - Swap the first and second nodes and link the new head to the result of the recursive call.

Recurrence Equation

- Let $T(n)$ be the time complexity of the algorithm for a list with n nodes.
- The recurrence relation can be written as:

$$T(n) = T(n-2) + O(1)$$

- This is because for each pair of nodes, the algorithm swaps them in constant time $O(1)$, and then recursively processes the remaining $n-2$ nodes.
- Base Case: If $n = 0$ or $n = 1$, no swap is needed, so $T(0) = O(1)$ and $T(1) = O(1)$.

Problem 3: Swap Every Two Adjacent Nodes

Solving the Recurrence

$$\begin{aligned} T(n) &= T(n-2) + O(1) \\ T(n-2) &= T(n-4) + O(1) \\ T(n-4) &= T(n-6) + O(1) \\ &\vdots \\ T(2) &= T(0) + O(1) \end{aligned}$$

Summing these gives:

$$T(n) = O(1) + O(1) + \cdots + O(1) \quad \text{(n/2 terms)}$$

Therefore, the time complexity is:

$$T(n) = O(n)$$

Problem 3: Swap Every Two Adjacent Nodes

Proof of Correctness: Induction

- **Base Case**: When $n = 0$ or $n = 1$, the algorithm correctly returns the head since no swaps are needed.
- **Inductive Hypothesis**: Assume the algorithm correctly swaps adjacent nodes for a list of $n - 2$ nodes.
- **Inductive Step**: For n nodes, the algorithm swaps the first two nodes, and then recursively calls itself to swap the next $n - 2$ nodes. By the inductive hypothesis, the recursive call correctly swaps the remaining nodes.
- Therefore, by mathematical induction, the algorithm correctly swaps adjacent nodes for any list.

The Sorting Problem

- **Input**: A sequence of n numbers $\{a_1, a_2, \ldots, a_n\}$.
- **Output**: A permutation (reordering) $\{a'_1, a'_2, \ldots, a'_n\}$ of the input sequence such that:

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

- The input is usually an n-element array but can also be represented in other ways, such as a linked list.

The Structure of the Data

- In practice, the numbers to be sorted are rarely isolated values. Each is part of a record.
- Each record contains:
  - **Key**: The value to be sorted.
  - **Satellite Data**: Additional information that must be permuted along with the key.
- When sorting, we often permute the records themselves, or to minimize data movement, we permute an array of pointers to the records.

Why Sorting?

Why is Sorting Important?

- **Inherent Need**: Many applications require sorted data, such as banks sorting checks by check number.
- **Subroutine in Algorithms**: Sorting is a key subroutine in many algorithms. For example, rendering graphical objects in the correct order may require sorting.
- **Rich Set of Techniques**: Sorting algorithms use various design techniques that are essential in other algorithms as well.
- **Historical and Theoretical Importance**: Sorting has been central to the study of algorithms for decades and helps develop core algorithmic concepts.

Sorting and Algorithm Design

- Sorting is often considered the most fundamental problem in algorithm design.
- Many key techniques for algorithm design appear in sorting algorithms developed over the years.
- Sorting helps introduce concepts such as divide and conquer, comparisons, recursion, and algorithmic efficiency.

Engineering Issues in Sorting

- The fastest sorting algorithm depends on various factors:
  - Prior knowledge about the keys and satellite data.
  - The memory hierarchy of the host computer (caches, virtual memory).
  - The software environment (how the sorting function interacts with other parts of the system).
- These issues are often best addressed at the algorithmic level rather than tweaking the implementation code.

Comparison-Based Sorting

- Sorting algorithms that determine order by comparing elements.
- Examples:
  - **Bubble Sort**: Repeatedly swapping adjacent elements if they are in the wrong order.
  - **Selection Sort**: Repeatedly selecting the smallest (or largest) element from the unsorted part of the list and placing it at the beginning.
  - **Insertion Sort**: Builds the sorted array one element at a time by repeatedly inserting the current element into the correct position in the sorted portion.
  - **Merge Sort**: Dividing the array in half, sorting each half, and merging the results.
  - **Quick Sort**: Selecting a pivot element, partitioning the array, and recursively sorting the partitions.
  - **Heap Sort**: Building a heap from the input data and then extracting the maximum (or minimum) element repeatedly.

Conclusion

- Sorting is a fundamental problem in computer science due to its wide range of applications.
- Sorting algorithms help illustrate essential algorithmic techniques such as recursion, divide-and-conquer, and comparison-based decision making.
- Understanding sorting is key to understanding the complexity and performance of many other algorithms.

Introduction to Bubble Sort

What is Bubble Sort?

- Bubble Sort is a simple comparison-based sorting algorithm.
- The algorithm repeatedly compares adjacent elements and swaps them if they are in the wrong order.
- After each pass through the list, the largest unsorted element "bubbles" to its correct position.

How Bubble Sort Works

Step-by-Step Process:

- Traverse the list multiple times, comparing adjacent elements.
- Swap adjacent elements if they are out of order.
- After each pass, the largest unsorted element is placed in its correct position.
- Repeat the process for the remaining unsorted part of the list.

Example of Bubble Sort (Pass 1)

Initial List: $[5, 1, 4, 2, 8]$

- Compare 5 and 1, swap: $[1, 5, 4, 2, 8]$
- Compare 5 and 4, swap: $[1, 4, 5, 2, 8]$
- Compare 5 and 2, swap: $[1, 4, 2, 5, 8]$
- Compare 5 and 8, no swap: $[1, 4, 2, 5, 8]$
- Largest element 8 is in its correct position.

Example of Bubble Sort (Pass 2)

Next List: $[1, 4, 2, 5, 8]$

- Compare 1 and 4, no swap.
- Compare 4 and 2, swap: $[1, 2, 4, 5, 8]$
- Compare 4 and 5, no swap.
- Largest element 5 is now in its correct position.

Example of Bubble Sort (Final Pass)

Next List: $[1, 2, 4, 5, 8]$

- Compare 1 and 2, no swap.
- Compare 2 and 4, no swap.
- No swaps made, the list is sorted!

Final Sorted List: $[1, 2, 4, 5, 8]$

Bubble Sort Algorithm

Algorithm BubbleSort(arr)

```
n ← length of arr
for i = 0 to n − 1 do                          ▷ Outer Loop
    for j = 0 to n − i − 1 do                  ▷ Inner Loop
        if arr[j] > arr[j + 1] then            ▷ Swap if out of order
            Swap arr[j] and arr[j + 1]
        end if
    end for
end for
```

Time Complexity of Bubble Sort

Time Complexity Analysis:

- **Worst-case time complexity**: $O(n^2)$ (when the list is in reverse order).
- **Best-case time complexity**: $O(n)$ (if the list is already sorted and we use early termination).
- **Average-case time complexity**: $O(n^2)$ for a randomly ordered list.

Space Complexity of Bubble Sort

Space Complexity:

- Bubble Sort is an **in-place sorting algorithm**, meaning it requires only a constant amount of additional memory.
- **Space Complexity**: $O(1)$, as it uses a fixed number of extra variables regardless of the input size.

Optimization: Early Exit

- We can optimize Bubble Sort by tracking whether any swaps were made during a pass.
- If no swaps are made, the list is already sorted, and we can terminate early.
- This reduces the number of unnecessary passes and improves performance for nearly sorted lists.

Optimized Bubble Sort Algorithm

Algorithm OptimizedBubbleSort(arr)

```
n ← len(arr)
for i = 0 → n − 1 do
    swapped ← False
    for j = 0 → n − i − 1 do
        if arr[j] > arr[j + 1] then
            swap(arr[j], arr[j + 1])
            swapped ← True
        end if
    end for
    if not swapped then
        break                ▷ Terminate if no swaps occurred
    end if
end for
```

Advantages and Disadvantages

Advantages of Bubble Sort:

- Simple and easy to understand.
- Requires no additional memory beyond the input array.

Disadvantages of Bubble Sort:

- Very inefficient for large datasets due to its $O(n^2)$ time complexity.
- Redundant comparisons and swaps, especially when the list is nearly sorted.

Conclusion

Summary of Bubble Sort:

- Bubble Sort is a simple but inefficient sorting algorithm, primarily used for small datasets or educational purposes.
- While it can be optimized with early termination, its $O(n^2)$ time complexity makes it impractical for large datasets.
- Despite its inefficiencies, Bubble Sort is an excellent tool for learning sorting concepts and basic algorithm design.