1. Suppose we perform a sequence of n operations on a data structure in which the i-th operations cost i if i is an exact power of 2 and 1 otherwise. Use both Aggregate analysys method and Amortized analysis method to determine the amortized cost per operation.

**Ans:**

**Amortized Analysis for Operations with Variable Costs**

**Problem Definition**

* Operation cost: i if i is an exact power of 2, and 1 otherwise
* Power of 2 operations occur at positions 1, 2, 4, 8, 16, ..., up to the largest power of 2 ≤ n

**Aggregate Analysis Method**

1. **Cost of power of 2 operations:**
   * Sum of 2⁰ + 2¹ + 2² + ... + 2ᵏ where 2ᵏ is largest power of 2 ≤ n
   * This sum equals 2ᵏ⁺¹ - 1
2. **Cost of non-power of 2 operations:**
   * Each costs 1
   * Number of such operations = n - (k+1)
   * Total cost = n - (k+1)
3. **Total cost calculation:**
   * T(n) = (2ᵏ⁺¹ - 1) + (n - (k+1))
   * T(n) = 2ᵏ⁺¹ + n - k - 2
4. **Amortized cost per operation:**
   * T(n)/n

**Potential Method Analysis**

1. **Potential function definition:**
   * Φ(Dᵢ) = 2ʲ where j is the largest power of 2 less than i
   * This function captures "stored energy" after the i-th operation
2. **Amortized cost calculation:**
   * Cᵢ = cᵢ + Φ(Dᵢ) - Φ(Dᵢ₋₁)
   * For power of 2 operations: high actual cost but moderate increase in potential
   * For non-power of 2 operations: low actual cost with small or zero potential change
3. **Result:**
   * The amortized cost balances out across all operations
   * Expensive power of 2 operations are offset by inexpensive non-power of 2 operations
   * Overall amortized cost is constant per operation
4. Improve the BubbleSort implementation so that in the best case (which means here that the input is already sorted), the algorithm runs in O(n) time. Explain why your new version works --- in other words, prove that the best case running time of your code is O(n). Call your new Java file BubbleSort1.java.

**Ans:**

A screenshot of a computer program

AI-generated content may be incorrect.

At the start of each outer loop iteration, the swapped flag is initialized to false to monitor whether any elements were exchanged during the inner loop. If a swap occurs, it signifies that some elements were out of order and needed rearrangement. However, if no swaps happen in an iteration, the algorithm terminates early, resulting in a best-case time complexity of O(n) for already sorted arrays.

1. Recall that in BubbleSort, at the end of the first pass through the outer loop, the largest element of the array is in its final sorted position. After the next pass, the next largest element is in its final sorted position. After the ith pass (i=0,1,2,…), the largest, second largest,…, i+1st largest elements are in their final sorted position. Use this observation to cut the running time of BubbleSort in half. Implement your solution in code, and prove that you have improved the running time in this way. Call your new Java file, which contains the improvements from this problem and the previous problem, BubbleSort2.java.

**Ans:**

A screenshot of a computer program

AI-generated content may be incorrect.

The optimized BubbleSort implementation improves efficiency by tracking the last position where a swap occurred (using the newn variable). This allows the algorithm to recognize that elements beyond this position are already sorted and don't need further comparisons.

For example, with an array like {3, 4, 6, 1, 19}, once the misplaced 1 is moved to its correct position in the array, subsequent passes only need to examine the portion of the array that might still contain unsorted elements. This modification significantly reduces the number of unnecessary comparisons, especially for arrays that are partially sorted, as the algorithm can progressively shrink its active sorting range based on actual swap activity.

1. An array A holds n integers, and all integers in A belong to the set {0,1, 2}. Describe an O(n) sorting algorithm for putting A in sorted order. Your algorithm may not make use of auxiliary storage such as arrays or hashtables (more precisely, the only additional space used, beyond the given array, is O(1)). Give an argument to explain why your algorithm runs in O(n)

**Ans:**

This algorithm implements a single-pass variant of Edsger Dijkstra's Dutch National Flag algorithm using three pointers: low, mid, and high. As mid scans through the array from left to right, each element is examined exactly once and placed in its correct position with at most one swap, with each of the three possible actions (for values 0, 1, and 2) taking constant time. Since we process each of the n elements with O(1) operations, the overall time complexity is O(n). The algorithm meets the O(1) extra space requirement by using only three index variables as additional storage. It works by maintaining four distinct regions in the array: elements from index 0 to (low-1) contain only 0s, from low to (mid-1) contain only 1s, from mid to high contain unprocessed elements, and from (high+1) to (n-1) contain only 2s. After completion, the array is fully sorted with all 0s first, followed by all 1s, and then all 2s.

A screenshot of a computer program

AI-generated content may be incorrect.