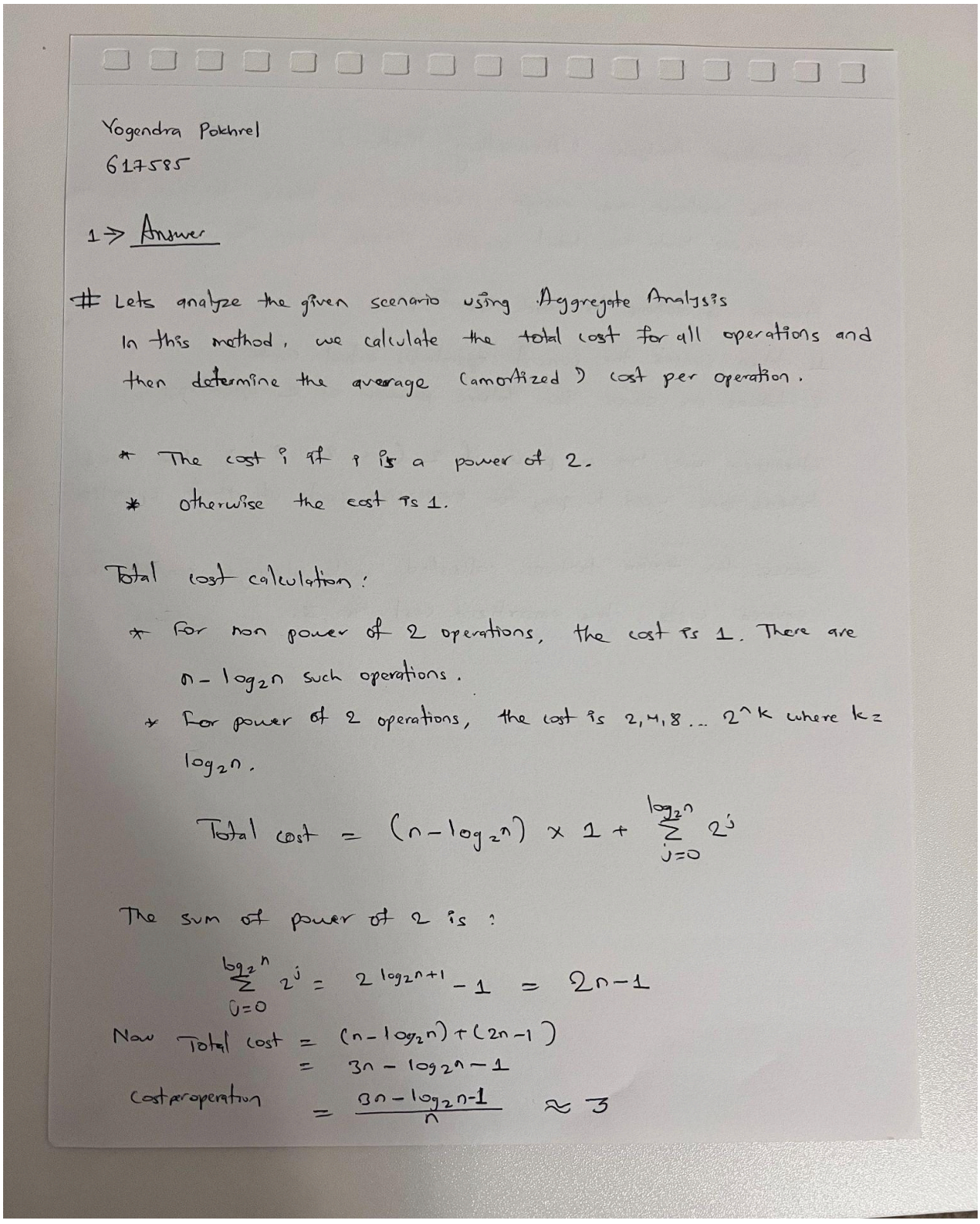
## Yogendra Pokhrel - 617585

### 1. Answer



Yogendra Pokhrel
617585

1 ⇒ Answer

# Lets analyze the given scenario using Aggregate Analysis
In this method, we calculate the total cost for all operations and then determine the average (amortized) cost per operation.

* The cost is if i is a power of 2.
* otherwise the cost is 1.

Total cost calculation:

* For non power of 2 operations, the cost is 1. There are $n - \log_2 n$ such operations.
* For power of 2 operations, the cost is 2, 4, 8 ... $2^k$ where $k = \log_2 n$.

$$\text{Total cost} = (n - \log_2 n) \times 1 + \sum_{j=0}^{\log_2 n} 2^j$$

The sum of power of 2 is:

$$\sum_{j=0}^{\log_2 n} 2^j = 2^{\log_2 n + 1} - 1 = 2n - 1$$

Now Total cost $= (n - \log_2 n) + (2n - 1)$
$= 3n - \log_2 n - 1$

Cost per operation $= \frac{3n - \log_2 n - 1}{n} \approx 3$

2) Amortized Analysis (Accounting Method)

In this method, we assign "credit" or "tokens" to each operation to ensure that the total credits always cover the actual costs.

Assign 3 tokens to each operation

1 token covers the current operations actual cost

2 tokens are saved for future power of 2 operations.

Whenever we hit a power of 2 (i.e 2, 4, 8 ..) the saved tokens are used to pay for expensive cost of that operation

Since the extra tokens collected will always cover the required costs, the amortized cost is 3.

## 2. Answer:

**Improved Bubble Sort Implementation in Java**
To improve the Bubble Sort for the best case scenario (already sorted array), we can introduce a flag that tracks whether any swaps occurred during a pass. If no swaps are made, the array is already sorted, and we can exit early.

```
public class BubbleSort1 {

  public static void bubbleSort(int[] arr) {
      int n = arr.length;
      boolean swapped;

      for (int i = 0; i < n - 1; i++) {
          swapped = false;

          for (int j = 0; j < n - i - 1; j++) {
              if (arr[j] > arr[j + 1]) {
                  int temp = arr[j];
                  arr[j] = arr[j + 1];
                  arr[j + 1] = temp;
                  swapped = true;
              }
          }

          if (!swapped) break;
      }
  }
}
```

**Explanation (Why It's O(n) in the Best Case)**
In the best case (already sorted array), no swaps will be needed. The outer loop will run once, and since swapped remains false, the algorithm breaks out early. Since only one pass through the array occurs, the best-case time complexity is O(n).

**Proof of Best-Case Complexity**
Outer loop runs once: O(1)
Inner loop iterates through n elements: O(n)
Best-case complexity = O(n)

## 3. Answer

**Enhanced Bubble Sort Implementation (BubbleSort2.java)**

To improve Bubble Sort further, we can reduce the number of comparisons in each pass. Since the largest element moves to its correct position in each iteration, we don't need to compare those sorted elements again.

By reducing the number of comparisons in each pass, we effectively cut the running time nearly in half.

```
public class BubbleSort2 {

  public static void bubbleSort(int[] arr) {
      int n = arr.length;
      boolean swapped;
      int lastUnsortedIndex = n - 1;

      while (lastUnsortedIndex > 0) {
          swapped = false;

          for (int j = 0; j < lastUnsortedIndex; j++) {
              if (arr[j] > arr[j + 1]) {
                  int temp = arr[j];
                  arr[j] = arr[j + 1];
                  arr[j + 1] = temp;
                  swapped = true;
              }
          }

          if (!swapped) break;
          lastUnsortedIndex--;
      }
  }
}
```

### Explanation (Key Changes for Efficiency)

lastUnsortedIndex tracks the boundary of the unsorted region.
➔ Each pass places the largest element in its correct position, so future passes exclude that element.
Early exit is still present to ensure O(n) performance in the best case.

## 4. Answer:

**Sorting an Array with Only {0, 1, 2} in O(n) Without Extra Space**

We can use the Dutch National Flag Algorithm (also known as the 3-way partitioning algorithm) to efficiently sort the array in O(n) time using O(1) extra space.

## Algorithm (Dutch National Flag Algorithm)

1. **Initialize Three Pointers:**
    - `low` → Tracks the boundary for `0`s.
    - `mid` → Current element under inspection.
    - `high` → Tracks the boundary for 2s.
2. **Traversal Logic:**
    - While `mid ≤ high`:
        - If `A[mid] == 0`:
          ➔ Swap `A[mid]` with `A[low]` and increment both `low` and `mid`.
        - If `A[mid] == 1`:
          ➔ Just increment `mid` (element 1 is already in the correct position).
        - If `A[mid] == 2`:
          ➔ Swap `A[mid]` with `A[high]` and decrement `high` (but don't increment `mid` since the swapped value may need to be checked).
3. The array will be sorted after one complete pass.

```
public class Sort1 {

  public static void sortColors(int[] A) {

      int low = 0, mid = 0, high = A.length - 1;



      while (mid <= high) {

          if (A[mid] == 0) {

              int temp = A[low];

              A[low] = A[mid];

              A[mid] = temp;

              low++;

              mid++;
```

```
        } else if (A[mid] == 1) {
            mid++;
        } else {
            int temp = A[mid];
            A[mid] = A[high];
            A[high] = temp;
            high--;
        }
    }
}
```

### Why Does This Algorithm Run in O(n)?

- Each element is visited once, either swapped or skipped.
- There's no nested loop, all operations are performed in a single pass.
- The algorithm efficiently partitions the array using constant space (`low`, `mid`, `high` pointers).