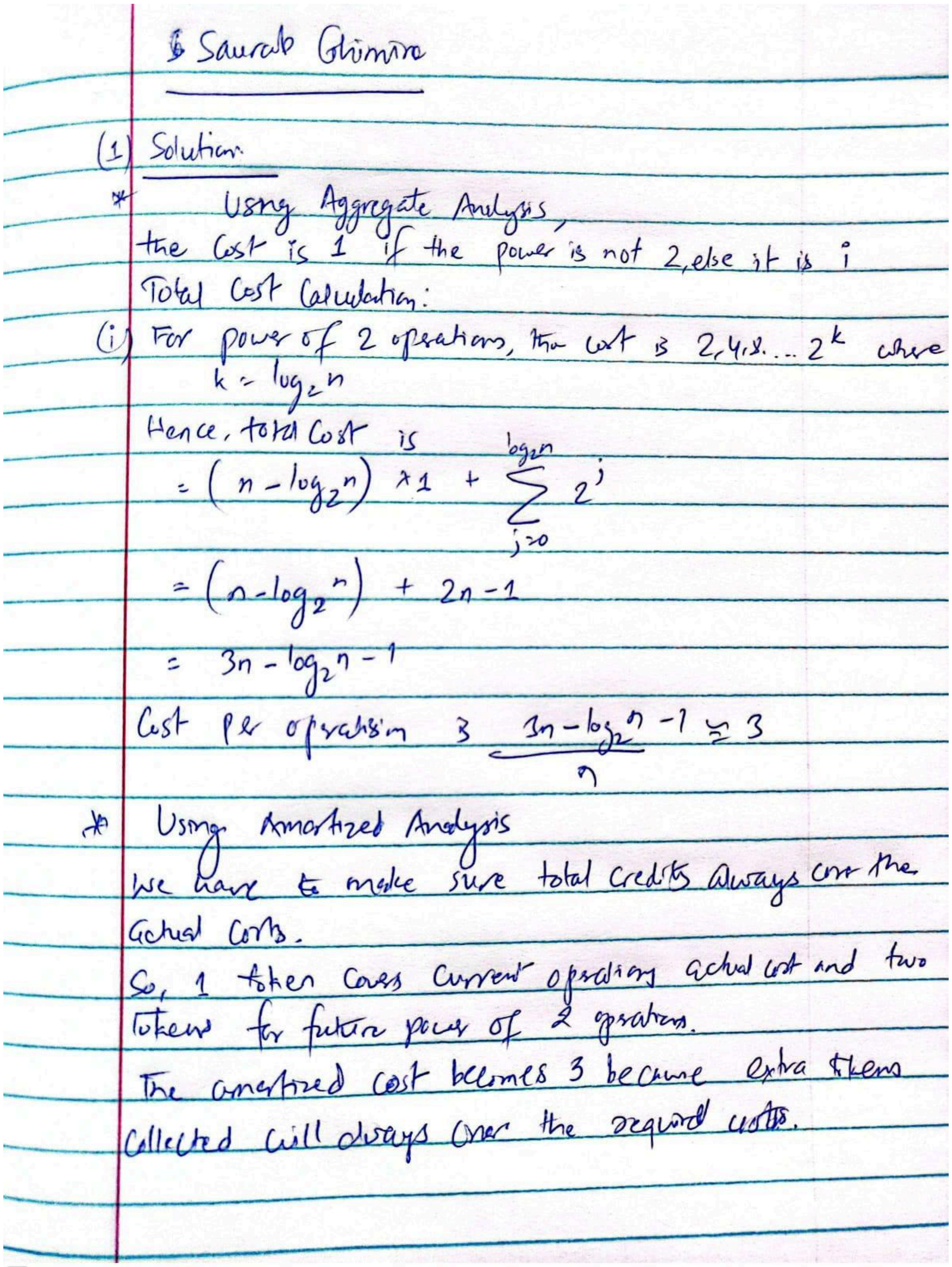
CS435DE - Lab 3

Saurab Ghimire
417555

Problem 1: Solution



Saurab Ghimire

(1) Solution

* Using Aggregate Analysis,

the cost is 1 if the power is not 2, else it is i

Total Cost Calculation:

(i) For power of 2 operations, the cost is 2, 4, 8 ... 2^k where $k = \log_2 n$

Hence, total cost is

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

$$= (n - \log_2 n) + 2n - 1$$

$$= 3n - \log_2 n - 1$$

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

* Using Amortized Analysis

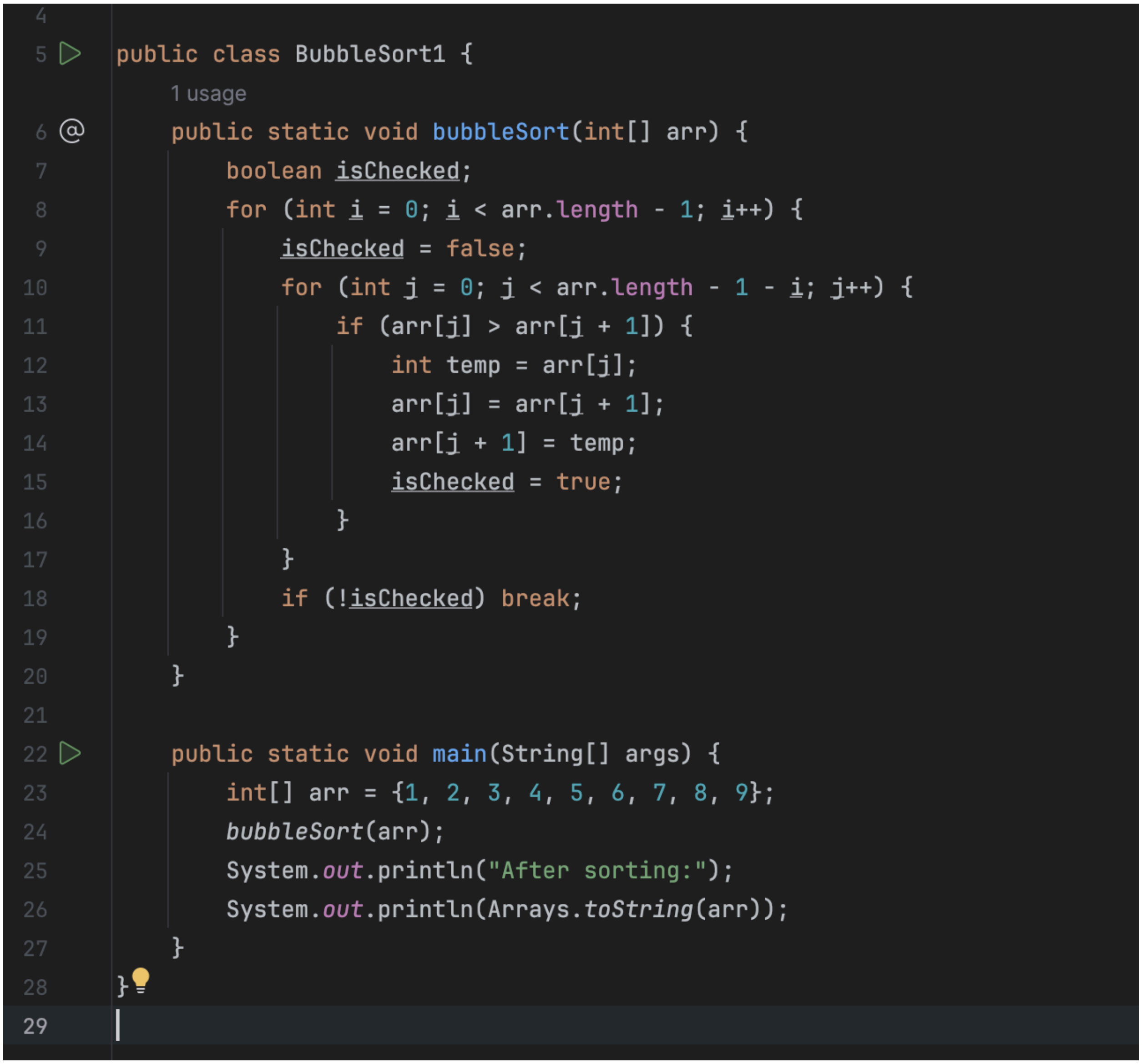
We have to make sure total credits always cover the actual costs.

So, 1 token covers current operation actual cost and two tokens for future power of 2 operations.

The amortized cost becomes 3 because extra tokens collected will always cover the required costs.

Problem 2: Solution

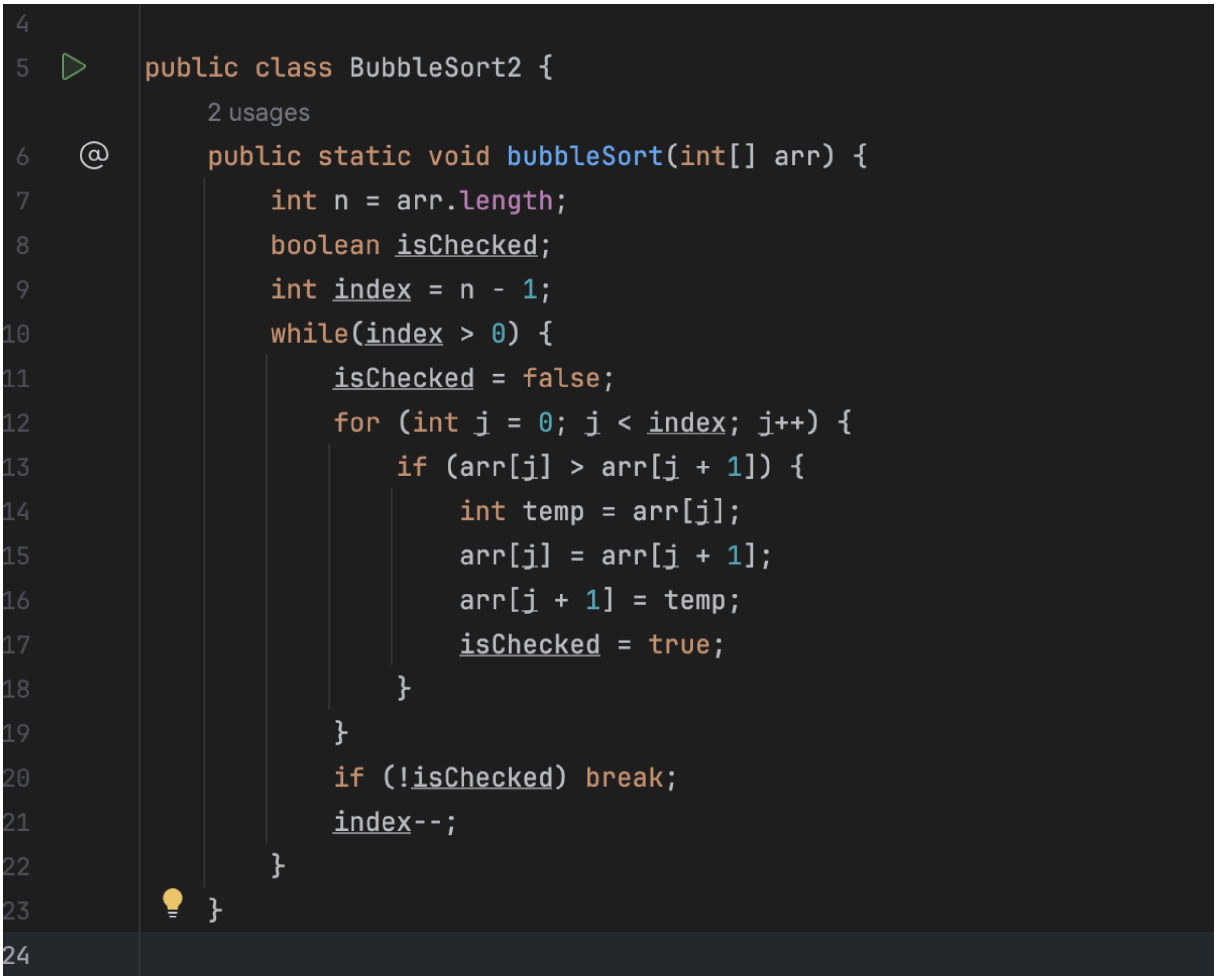
Bubble sort implementation can be improved for the best case scenario with sorted array by adding a variable of type boolean to check if any swapping is needed, so that we can break the for loop if it is already sorted. The outer loop will run exactly once if the array is already sorted. So the best case time complexity is O(n).



```java

public class BubbleSort1 {
    public static void bubbleSort(int[] arr) {
        boolean isChecked;
        for (int i = 0; i < arr.length - 1; i++) {
            isChecked = false;
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    isChecked = true;
                }
            }
            if (!isChecked) break;
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        bubbleSort(arr);
        System.out.println("After sorting:");
        System.out.println(Arrays.toString(arr));
    }
}

```

Problem 3: Solution

We know that after the ith pass (i=0,1,2,...) the (largest, second largest…,i+1st largest) elements are in the final sorted position, we can remove comparing the sorted elements once we have already done that. By reducing comparisons, we can reduce the time by 50 percentage. The early break is present to ensure O(n) performance in the best case scenario.



```java

public class BubbleSort2 {
    2 usages
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        boolean isChecked;
        int index = n - 1;
        while(index > 0) {
            isChecked = false;
            for (int j = 0; j < index; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    isChecked = true;
                }
            }
            if (!isChecked) break;
            index--;
        }
    }

```

Problem 4:
To sort an array A, that holds n integers, and all integers in A belong to the set {0,1,2} in O(n) time we can use Dutch National Flag Algorithm

```
sort(A)
   Initialize:
      low ← 0
      mid ← 0
      high ← length(A) - 1

   While mid ≤ high:
      If A[mid] = 0:
         Swap A[mid] and A[low]
         Increment low
         Increment mid
      Else If A[mid] = 1:
         Increment mid
      Else:
         Swap A[mid] and A[high]
         Decrement high
   End While
End Procedure

Procedure swap(A, i, j)
   temp ← A[i]
   A[i] ← A[j]
   A[j] ← temp
End Procedure
```

The algorithm runs in O(n) time because each element in the array is processed at most once. The `mid` pointer moves forward through the array, checking each element a single time. The `low` and `high` pointers also move, but they never revisit elements once they are placed correctly. Since swapping and comparisons both take constant time O(1), the overall complexity remains O(n).