1. Compute the number of primitive operations for each of the following algorithm fragments.
   1. Algorithm | # operations

sum ← 0 | 1

**for** i ← 0 **to** n-1 **do** | N - 1

sum ← sum + 1 | 3 \* N

**Total:**  **4 \* N**

* 1. Algorithm | # operations

sum ← 0 | 1

**for** i ← 0 **to** n-1 **do** | N - 1

**for** j ← 0 **to** n-1 **do** | (N - 1) \* (N - 1)

sum ← sum + 1 | 3\*(N - 1) \* (N - 1)

**Total: 4 \* (N – 1) \* (N – 1) + N**

1. The first for loop is not nested which is O(N). The second that consists of a nested for loop which is O(N2). Asymptotic running time: **O(N2) + O(N) is** **O(N2)**.

int[] arrays(int n) {

int[] arr = new int[n];

for(int i = 0; i < n; ++i){ // First loop

arr[i] = 1;

}

for(int i = 0; i < n; ++i) {

for(int j = i; j < n; ++j){ // Second loop

arr[i] += arr[j] + i + j;

}

}

return arr;

}

1. The Pseudo-Code for the *Merge* function:

**Algorithm** merge(arr1, arr2)

**Input:** an array arr1, arr2 of sorted integers

**Output:** an array of sorted integers consists of arr1, and arr2 elements

n ← arr1.length + arr2.length

result ← new int[n]

i1 ← 0

i2 ← 0

**for** i←0 **to** n-1 **do**

**if** i1 = arr1.length || arr1[i1] > arr2[i2] **then**

result[i] = arr2[i2]

i2 ++

**else**

result[i] = arr1[i1]

i1 ++

**return** result

The asymptotic running time of this algorithm is O(N) since we have only one for loop with O(N) runtime and an array initializing with O(N) runtime then **O(N) + O(N) = O(N).**

1. (A). The running time of the *removeDups* is O(N2) because M.contains(x) searches the whole **not sorted** list to lookup the value x which is O(N) and it is inside a for loop which makes it a nested operation. Hence, the runtime of the *removeDups* is O(N2).

(B). The following Pseudo-Code shows the improved O(N) removeDups algorithm.

**Algorithm** removeDups(L)

**Input:** a list L

**Output:** a list M containing the distinct elements of L

M ← new list

H ← new HashMap

**for** i←0 **to** L.size()-1 **do**

**if not** H.containsKey(L[i]) **then**

M.add(L[i])

H.put(L[i])

**return** M

The improved algorithm uses separate HashMap to mark the seen values and instead of M.contains(x) I used H.contiainsKey(x) which runs a constant time (O(1)) that makes the for loop not nested. Hence the running time of the algorithm is **O(N)**.

(C). I(i): M=[distinct elements of L1, L2, …, Li], H=[keys of distinct elements of L1, L2, …, Li ]

Base case: I(0): M=[L0], H=[L0] it is obviously correct;

Induction: Lets assume I(i) is correct then prove I(i + 1) is correct.

Proof: Lets define Mi as a state of M in the I(i), Hi as same.

1. If the new element Li + 1 is unique element that had no duplicate element before.

Then Hi.containKey() will result false and our if condition will be true. So, we add Li + 1 to the M and put into the H, that results M = Mi + Li + 1 , and H = Hi + Li + 1 .

Since, we defined M, and H the distinct elements of L1, L2, …, Li+1 and Li + 1 is unique element that had no duplicate element M = Mi + Li + 1 , and H = Hi + Li + 1  will satisfy the condition.

1. If the new element Li + 1 is not unique element that had a duplicate element before.

Then Hi.containKey() will result true and our if condition will be false. So, we left the M and H as is.

Since, we defined Li + 1 had a duplicate element then not changed M, and H will satisfy the condition.