Problem 1

Description: We were given the problem in which we were to sort an array of integers of size n that can have largest values of up to n^2 and sort it in O(n). To solve this problem, I used the radix sort technique in which we sorted the elements of the array by using counting sort according to their least significant digits one at a time and then moving upto the most significant digit. After sorting I summed up indices of all the values divisible by 3 and displayed the output.

Pseudo Code:

initializeElements(array, input)

    for i to length of input:

        array [i] = input[i]

CountSort(input, size, base):

    sorted=[size] empty array of size:size

    i=0

    count = [10] empty array to store the counts of elements inside it

    initialize the elements of count to 0

    for i from 0 to size:

        index= (input[i] /base) %10 // calculating the value of the digit at the base position

        count[index] ++ // increasing the count of the element at the index

    calculating the cumulative sum of the elements of the array from the start

    for i from 0 to 10:

        count[i] += count[i-1]

    adding the values to the sorted array

    for i from size -1 to 0:

        index= (input[i] /base) %10 // calculating the value of the digit at the base position

        count[index] - -

        sorted[count[index]] = input[i]

    adding the sorted values back to the original input array

    for i from 0 to size:

        input[i]=sorted[i]

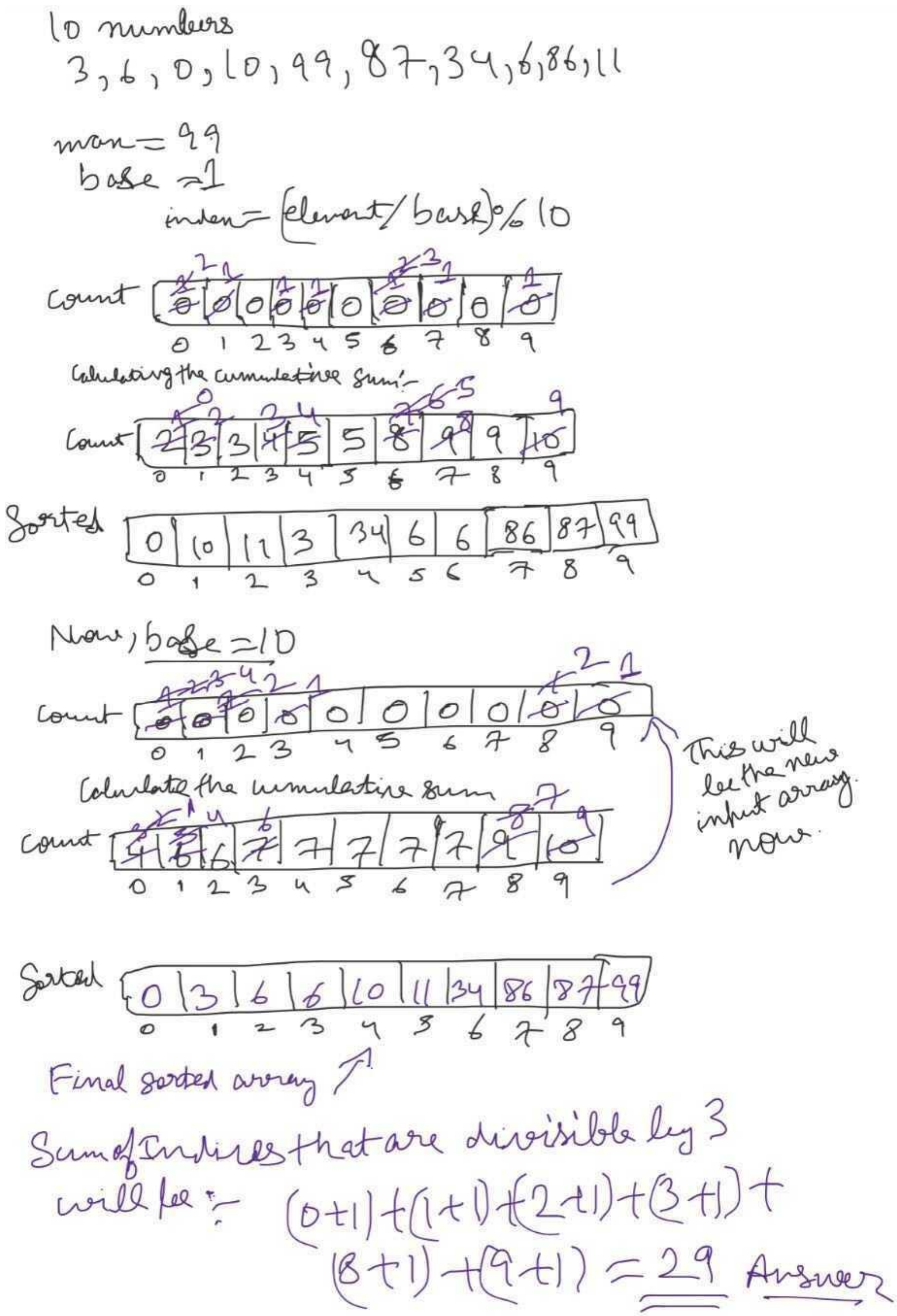
RadixSort(input, size):

```
    Calculating the element with the maximum value in input: max
     base=1
    calling the countsort for each digit of the maximum number in the input
    while(max/base>0):
        countSort(input, size, base)
        base=base*10
```

Main:

```
    Take standard input using scanner
    InitializeElements(input, line2)
    RadixSort(input, size)
    For i from 0 to size
        if input[i] is divisible by 3:
            sum+=i+1
    print (sum)
```

Sketch:



+1 because it starts from 0 when calculating the indices

Running Time estimate:

- Taking the standard input and adding the elements to the new array is O(n)
- Running the counting sort in the radix sort function for constant time O(d) d is a constant which is equal to the digits of the maximum value in the input array.
- Running the counting sort:
    - Increasing the count in the count array is O(n)
    - Going from size-1 to 0 and computing the sorted output is again O(n)
    - Adding the computed output to the initial input array which is again O(n)
- After sorting, going through all the elements and calculating the sum of indices is again O(n)
- So, the total complexity of the program will be:
  T(n)= O(n)+d*O(n)++d*O(n)+d*O(n)+O(n)=(d+3) *O(n)
  d+3 is a constant so the total asymptomatic complexity of the program will be O(n)

Problem 2

(a)

```
WHATDOIDO(integer left, integer right):
    if left==right:
        if A[left]<0 return (0, 0, 0, A[left])
        else return (A[left], A[left], A[left], A[left])        } O(1)
    if left<right:
        m = (left+right)/2 (rounded down)
        (lmaxsum, llmaxsum, lrmaxsum, lsum) = WHATDOIDO(left, m)        } O(n/2)
        (rmaxsum, rlmaxsum, rrmaxsum, rsum) = WHATDOIDO(m+1, right)     } O(n/2)
        maxsum = max{lmaxsum, rmaxsum, lrmaxsum+rlmaxsum}
        leftalignedmaxsum = max{llmaxsum, lsum+rlmaxsum}
        rightalignedmaxsum = max{rrmaxsum, lrmaxsum+rsum}               } O(1)
        sum = lsum+rsum
        return (maxsum, leftalignedmaxsum, rightalignedmaxsum, sum)
```

The recurrence of T(n) that captures the running time of the algorithm as closely as possible:

T(n)= 2 T(n/2) + d (d represents the complexity of a constant time operation)

(b)

Finding a tight bound for

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + d$

$$T(n) \le \begin{cases} d_0 & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + d & n > 1 \end{cases}$$

Telescoping:

$T(n) \le 2T\left(\frac{n}{2}\right) + d.$

$\le 2\left(2T\left(\frac{n}{4}\right) + d\right) + d = 4T\left(\frac{n}{4}\right) + 2d$

$\le 4\left(2T\left(\frac{n}{8}\right) + d\right) + 2d = 8T\left(\frac{n}{8}\right) + 3d$

$\vdots$

$\le 2^k T\left(\frac{n}{2^k}\right) + kd$ Base case $\frac{n}{2^k} = 1$, $k = \log_2 n$

$\le 2^{\log_2 n} T(1) + \log_2 n \cdot d.$

$= n d_0 + \log_2 n\, d.$

$\therefore T(n) \leq n d_0 + \log_2 n \, d. \quad d_2 = \max\{d_0, d\}$

So, $T(n) = n d_2 + \log_2 n \, d_2$

$= d_2 (n + \log_2 n).$

$= d_2(n + \log_2 n)$

$T(n) = O(n) \quad c = d_2 \ \forall n \geq 2$

Also, we get the same T(n)=O(n) using the master theorem.

(c)

- For an input A and integers left and right, the meaning of the variables is as follows:
    - maxsum: This represents the maximum sum of all the integers in the array A, from left to the right element index of the array. The sum does not contain the negative integers and replaces them with 0.
    - leftalignedmaxsum: It is described as the maximum value out of (llmaxsum and lsum+rlmaxsum),
        - where llmaxsum is the left-left max sum, that is maximum sum of all the positive integers on the left half of the left half of the input array A.
        - lsum is the sum of all the integers of the left half of the array A and rlmaxsum is the sum of all the +ve integers of the left half of the right half of the array A.
    - rightalignedsum: It is described as the maximum value out of (rrmaxsum and lrmaxsum+rsum)
        - where rrmaxsum is the right-right max sum, that is the maximum sum of only the +ve integers on the right half of the right half of the input array A.
        - lrmaxsum represents the left-right max sum, which is the sum of all the +ve integers on the right half of the left half of the elements of the array A. rsum is the sum of all the elements on the right half of the input array A.
    - sum: Sum contains the total sum of all the elements both +ve and -ve of the array.

- WHATDOIDO(1, n). After running this, the first value out of the 4 values will give the sum of all the positive integers in the array A. It will be the maximum sum of the array and it will not include the negative integers. If the array contains a negative integer, their value will be summed up as 0. For examples, if we have A as: [10,3,-1,-6,2,3]. The maxsum will be 10+3+0+0+2+3=18.

Problem 3

Description: To solve the given problem, I use the merge sort algorithm and along with counting the inversions using that, I multiply each and every weighted inversion and return the sum for each small problem inversion that I find. After that I am recursively adding all the sums together to get the total sum of all the inversions multiplied by each other which is the final solution.

PseudoCode:

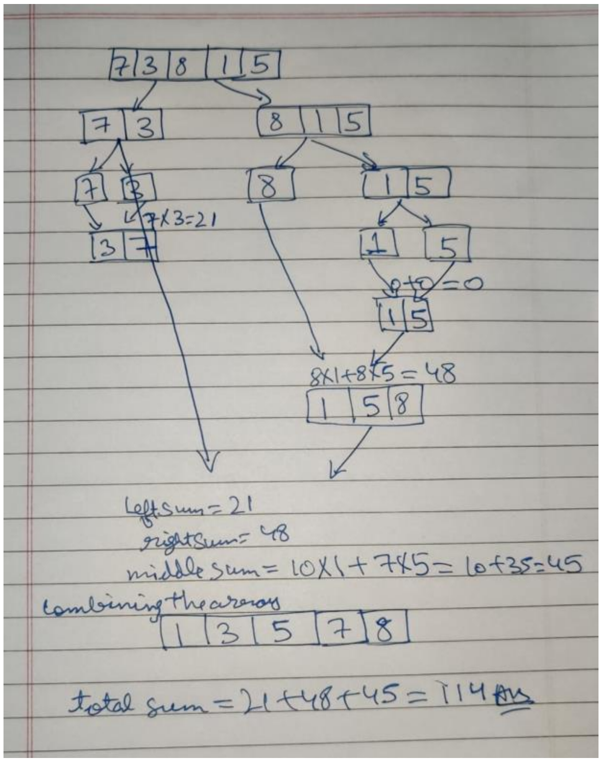
```
MergeSortAndCount(long[ ] input):
    m=input.length/2
    if(m==0) return 0
    leftArray= new array of size[m]
    rightArray = new array of size[input.length-m]
    copyElements(input, leftArray, 0, m-1)
    copyElements(input, rightArray, m, input.length-1)
    sumLeft= MergeSortAndCount(leftArray)
    sumRight= MergeSortAndCount(rightArray)
    sumMiddle= MergeAndCount(leftArray, rightArray, input)
    return sumLeft+sumRight+sumMiddle


copyElements(originalArray, newArray, start, end):
    int k=0
    for i in loop from start to end:
        newArray[k] = original[i]
        k++
```

```
MergeAndCount( leftArray, rightArray, inputArray):
    sum=0
    i=0.  j=0, k=0
    n=leftArray.length
    list= new Array of size [n+1]
    Initialize all the elements of the new list array to 0
    // the list will contain the cumulative sum of the left array in the reverse order
    for (i=n-1; i>=0;i--)
        // adding the cumulative sums of the elements into list in the reverse order
        list[ i ] = list[ i +1]+leftArray[ i ]
    i=0
    while i<leftArray.length and j<rightArray.length
        if(leftArray[i]<= rightArray[j]:
            input[k]=leftArray[i]
            k++, i++
        else
            input[ k ] = rightArray[ j ]
            // calculating the sum for the middle inversions
            sum+= list[ i ] * rightArray[ j ]
            k++, j++
    append all the remaining elements to the left Array and the right Array
    return sum

initializeElements( newArray, input )
    for i to length of input:
        newArray [ i ] = input[ i ]
MAIN:
    Take input
    InitializeElements( newArray, input)
    Print(mergeSortAndCount(newArray))
```

Sketch:



Running Time estimate:

- Taking the input from the array is O(n)
- Then in the recursive function creating 2 new arrays of the combined size n and copying n elements in O(n)
- The sumLeft and sumRight calculation is a recursive call for which the depth of the recursive tree is log(n)
- The mergeAndCount function which merges all the elements by comparing them and then returns the sum of the weighted inversions, the complexity for that will be O(n)
- Each recursive call for the tree is of the order O(n) and it is repeated log n times down the tree, so the total complexity of the program would be given by O(n log n)

O(n)= O(n) + O(n*log(n)) + O(n*log(n)) + d (d is a constant)

O(n)= O(n) + 2 *O(n*log(n)) + d (d is a constant)

So, the total complexity will be given by the dominating term which is O(n log n)

Problem 4

Problem 4. Master theorem: $T(n) = aT(\frac{n}{b}) + f(n)$

1. $T(n) = 3T(n/2) + n^2$

According to the master theorem, here

$a = 3$ $\quad \log_b a = \log_2 3$

$b = 2$

$K = 2$

here $K > \log_b a$ because $2 > \log_2 3$

~~Sor~~ So, it is case 3

and $T(n) = \Theta(n^2)$.

$T(n) = \Theta(n^2)$

2. $T(n) = 5\cancel{b}T(n/2) + n\log n$.

We need to apply the general form of master theorem which is:-

$T(n) = a\,T(\frac{n}{b}) + f(n)$ where $f(n) = \Theta(n^k \log^p n)$

$a = 5$ $\quad \log_b a = \log_2 5$

$b = 2$ $\quad \log_2 5 > k = 1$

$K = 1$ $\quad$ so, $T(n) = \Theta(n^{\log_b a})$

$p = 1$ $\quad = \Theta(n^{\log_2 5})$ Answer

3. $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$

The above expression is in the form of :-

$T(n) = aT\left(\frac{n}{b}\right) + n^K$

here :- $a = 2$ $\log_b a = \log_4 2$
$b = 4$
$K = 1/2$

here $\frac{1}{2} = \log_4 2$ $K = \log_b a$

So, this is Case 2.

$T(n) = \Theta(n^{\log_b a} \cdot \log n)$

$= \Theta(\sqrt{n} \log n)$.