Problem 1:

a) T(n)= $\Theta(n^2)$.
   Explanation: T(n)=T(n-2)+n=T(n-4)+(n-2)+n=T(n-6)+(n-4)+(n-2)+n stop at T(2)=0, so, T(n)=(n-n)+...+(n-2)+(n-0)=$\Theta(n^2)$.

b) T(n)= $\Theta(n^3)$.
   Explanation: Master method, a=4,b=2,f(n)=n^3, $\log_2 4$=2, $n^{\log_2 4}$=n^2,f(n)=$\Omega(n^{2+\varepsilon})$.
   Case 3: regularity: af(n/b)=4($(n/2)^3$)=$n^3/2 \le$cf(n) for c=1/2, so T(n)= $\Theta(n^3)$.

c) T(n)= $\Theta(n^2\ lg\ n)$.
   Explanation: Master method, a=9,b=3, f(n)=n^2, $\log_3 9$ =2, n^2=f(n),so
   Case 2: T(n)= $\Theta(n^2\ lg\ n)$.

Problem 2:

The recurrence is T(n)=4T(n/3)+n. T(n)= $\Theta(n^{\log_3 4})$.
Explanation: assume T(1)=0,Master method, a=4,b=3,f(n)=n, $\log_3 4$>1,
f(n)=n=O($n^{\log_3 4-\varepsilon}$),ε>0, case 1, So the runtime T(n)= $\Theta(n^{\log_3 4})$.

Problem 3:

a) Write pseudo-code a recursive ternary search algorithm.

```
function TernarySearch(key, arr, left, right)
  // empty subarray: not found
  if left > right
    return false

  // single-element subarray: found if match, not found otherwise
  if left == right
    return key == arr[left]

  // if key is less than min or greater than max: not found
  if (key < arr[left]) or (key > arr[right])
    return false

  // calculate midpoints
  mid1 = left + floor((right - left) / 3)
  mid2 = left + 2 * floor((right - left) / 3) + 1

  // check key at midpoints
  if (key == arr[mid1]) or (key == arr[mid2])
    return true

  if key < arr[mid1]
    // search lower third
    return TernarySearch(key, arr, left, mid1 - 1)
  else if key > arr[mid2]
    // search upper third
```

```
    return TernarySearch(key, arr, mid2 + 1, right)
  else
    // search middle third
    return TernarySearch(key, arr, mid1 + 1, mid2 - 1)
}
```

b) Because at each iteration, ternary search makes at most 4 comparations, it's constant time. So, the time complexity of it should be: T(n)=T(n/3)+4
c) f(n)=4*n^0, a=1,b=3, $\log_3 1$ =0, master method, case 2: T(n)= $n^0 \log_3 n$= $\Theta(\log_3 n)$.

Problem 4:

a) //Mergesort3 algorithm

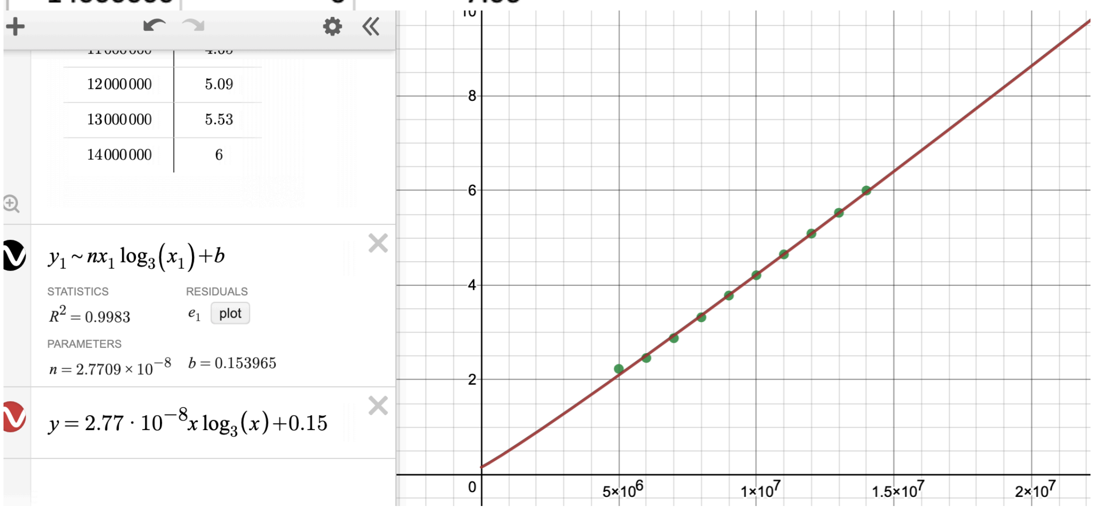
```
void Mergesort3(original[],low, high, aim[]){
        //the end of the loop
        if(high-low<2) return;

        mid1=low+(high-low)/3;
        mid2=low+2*(high-low)/3+1;
        //divide into 3 parts
        mergesort3(aim,low,mid1, original);
        mergesort3(aim,mid1,mid2, original);
        mergesort3(aim,mid2,high, original);
        //merge sorted original into aim
        merge3(aim,low,mid1,mid2,high,original)
}
//Merge3 function
void merge3(original[],low,mid1,mid2,high,aim[]){
        while (has 3n elements)
          {aim[index++]=original.smallest[index++];}
        while(has 2 elements left)
        { //1&3parts,2&3parts,1&3parts.
          aim[index++]=original.smaller[index++];}
        while(has 1 element)
        {aim[index++]=original[index++];}
}
```

b) T(n)=3T(n/3)+n.
Explanation: since it divides into 3 parts, it calls itself 3 times and merge 1 time. So the merging takes n times.
c) T(n)=n $\log_3 n$.
Explanation: a=3,b=3, $\log_3 3$=1, following the master method, it's case 2: T(n)=n $\log_3 n$.

Problem 5:

a) in teach
b) in teach
c) the table of running times for merge3 and merge2 algorithm is following below:

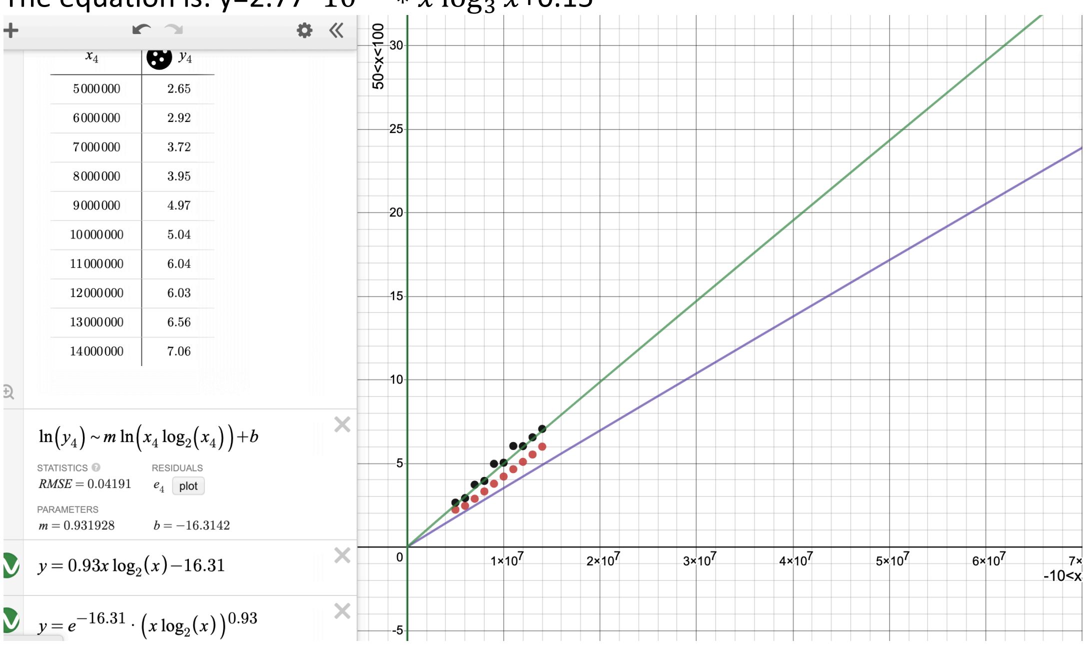
n	merge3	merge2
5000000	2.23	2.65
6000000	2.46	2.92
7000000	2.88	3.72
8000000	3.32	3.95
9000000	3.78	4.97
10000000	4.21	5.04
11000000	4.65	6.04
12000000	5.09	6.03
13000000	5.53	6.56
14000000	6	7.06



d)

As it shows, nlog$_3$ n best fits each data set.

The equation is: y=2.77*10^{-8} $* \; x \log_3 x$+0.15



e)

The purple one is Mergesort3, the green one is Mergesort. The Mergesort runs faster. My experiment is almost the same result as theory. The complexity of mergesort is almost: $\mathrm{nlog}_2 n$, and the mergesort3 is: $\mathrm{nlog}_3 n$. In theory, the mergesort3 is a little faster than mergesort2.