Lab 3

Houssam Eddine ATIF

N# 610165

1/

A/ Goofy's sorting procedure will work if the array is already sorted, so this procedure will return the array in the first step.

B/ the best case is when the array is already sorted.

C/ the running time of the best case is O(n), because he will check an array of n value.

D/ the worst case is infinity because he will arrange the number randomly in the table.

E/ This algorithm is not inversion-bound, because in the step two this algorithm arrange randomly the array without any comparison so the number of comparisons could be less than the number of inversion which mean that it's not inversion-bounded.

2/

```
public static int[] sort(int[] t) {

		int count0=0;
		int count1=0;

		for(int i=0;i<t.length;i++) {
			if(t[i]==0) count0++;
			else if(t[i]==1) count1++;
		}

		for(int i=0;i<t.length;i++) {
			if(i+1<=count0) t[i]=0;
			else if(i+1<=count1+count0 && i+1>count0) t[i]=1;
			else t[i]=2;
		}

		return t;
	}
```

Run time of the algorithm is $\Theta(n)$ because we have two loops each loop has runtime equal to $\Theta(n)$

So $\Theta(n)+\Theta(n)=\Theta(n)$.

3/ Bubble Sort

a/

```
private void bubbleSort(){

		int len = arr.length;
		boolean sorted;
		for(int i = 0; i < len; ++i) {
			sorted=true;
			for(int j = 0; j < len-1; ++j) {
				if(arr[j]> arr[j+1]){
					sorted=false;
					swap(j,j+1);
				}
			}
			if(sorted==true) break;
		}
	}
```

b/

```
private void bubbleSort(){

		int len = arr.length;
		boolean sorted;
		for(int i = 0; i < len; ++i) {
			sorted=true;
			for(int j = 0; j < len-1-i; ++j) {
				if(arr[j]> arr[j+1]){
					sorted=false;
					swap(j,j+1);
				}
			}
			if(sorted==true) break;
		}
	}
```

c/

- **What are the results?**

```
61 ms -> InsertionSort
111 ms -> SelectionSort
290 ms -> BubbleSort2
350 ms -> BubbleSort1
491 ms -> BubbleSort
```

- **Are the results what you expected? Explain why the running times turned out the way they did.**
  1/ For **bubbleSort1** it's normal to have run time less than bubbleSort because we optimize the outer loop. So, if the array is sorted after some runs of the outer loop, our program will stop immediately, we try to make this solution by saving the state (sorted or not) in a new variable and test it after each time we go out from the inner loop.
  2/ For bubbleSort2 we optimize the inner loop of bubbleSort1. In this sorting algorithm (bubbleSort) we move the larger value to the right of the table, so we don't have to test every time the sorted value in the right side of the array, so we shorted the inner loop after every iteration of the outer loop, the result cut the running time in half, that's why it's more faster.

4/

```
public static int count01(int[] a, int i, int j,int n) {

                  if(a[i]==1) {
                     System.out.println("number of 1: " + a.length + " number of 0:
0");
                     return 0;
                  }
                  else if(a[j]==0) {
                     System.out.println("number of 1: 0" + " number of 0: "+
a.length);
                     return 0;
                  }
                    if(a[(i+j+1)/2]==0 && a[((i+j+1)/2)+1]==1){

                           System.out.println("number of 1: "+ (a.length-
((i+j+1)/2+1)) + " number of 0: "+ ((i+j+1)/2+1));
                     return 0;
                    }
                    if(a[(i+j+1)/2]==1 && a[((i+j+1)/2)-1]==0) {
                           System.out.println("number of 1: "+ (a.length-((i+j+1)/2) +
" number of 0: "+ ((i+j+1)/2)));
                           return 0;
                    }
                    if(a[(i+j+1)/2]==0 && a[((i+j+1)/2)+1]==0){
                           n++;
                           return count01(a,(i+j+1)/2,j,n);
                    }
                    else{
                           System.out.println("-->3");
```

```
                    n++;
                    return count01(a,i,(i+j+1)/2,n);
                }
    }
```

This algorithm work with dichotomy method we divide the array into two, we test if the middle value of the array:

- **Case 1:** equal 0 and the next value is 0, we divide another time the right side of the table until we find the value 1;
- **Case 2:** equal 1 and the previous value is 1, we divide another time the left side of the table until we find the value 0;

This algorithm works recursively by dividing in each step the array in two.

This is the steps of each self-call: a/2, a/4, a/8 … $a/2^n$ (2^n is the length of the array)

- ⇨ n is the number of self-calls
- ⇨ and the number of steps in the worst case is n;
- ⇨ n=log(a.length)
- ⇨ O(Log(n)) witch is o(n)