

## LAB 2: PART 2.1

### EXERCISE 1

#### a. ARRAY A

3	5	7	6	9	8	12
---	---	---	---	---	---	----

##### Pass 1

3	5	7	6	9	8	12	Compare 3,5. No swap
3	5	7	6	9	8	12	Compare 5,7. No swap
3	5	6	7	9	8	12	Compare 7,6. Swap occurred since $7 > 6$
3	5	6	7	9	8	12	Compare 7,9. No swap
3	5	6	7	8	9	12	Compare 9,8. Swap occurred since $9 > 8$
3	5	6	7	8	9	12	Compare 9,12. No swap
Total no of swap = 2, listSize = 6							

\*note: rest of the pass is the same since the array has been sorted.

#### ARRAY B

8	9	6	7	5	3	12
---	---	---	---	---	---	----

##### Pass 1

8	9	6	7	5	3	12	Compare 8,9. No swap
8	6	9	7	5	3	12	Compare 9,6. Swap ( $6 < 9$ )
8	6	7	9	5	3	12	Compare 9,7. Swap ( $7 < 9$ )
8	6	7	5	9	3	12	Compare 9,5. Swap ( $5 < 9$ )
8	6	7	5	3	9	12	Compare 9,3. Swap ( $3 < 9$ )
8	6	7	5	3	9	12	Compare 9,12. No swap
Total no of swap = 4, listSize = 6							

##### Pass 2

6	8	7	5	3	9	12	Compare 8,6. Swap ( $6 < 8$ )
6	7	8	5	3	9	12	Compare 8,7. Swap ( $7 < 8$ )
6	7	5	8	3	9	12	Compare 8,5. Swap ( $5 < 8$ )
6	7	5	3	8	9	12	Compare 8,3. Swap ( $3 < 8$ )
Total no of swap = 4, listSize = 6							

##### Pass 3

6	7	3	5	8	9	12	Compare 5,3. Swap ( $3 < 5$ )
Total no of swap = 1, listSize = 6							

##### Pass 4

6	3	7	5	8	9	12	Compare 7,3. Swap ( $3 < 7$ )
6	3	5	7	8	9	12	Compare 7,5. Swap ( $5 < 7$ )
Total no of swap = 1, listSize = 6							

##### Pass 5

3	6	5	7	8	9	12	Compare 6,3. Swap ( $3 < 6$ )
3	5	6	7	8	9	12	Compare 6,5. Swap ( $5 < 6$ )
Total no of swap = 1, listSize = 6							

Total swaps = 11, Phase=5

Array A has better efficiency compared to Array B.

b. The ArrayB is not efficient since it is the worst case for the sorting. Hence, the program can be improved by implementing condition that check if the list sorted should be added at the external loop.

c.

```
//NAME: BRENDAN DYLAN GAMPA ANAK JOSEPH DUSIT@DUSIT
//MATRIC NO: A20EC0021

#include <iostream>
using namespace std;

void BubbleSort(int data[], int listSize){
    int pass, tempValue;
    bool sort=false;
    for(pass=1;(pass<listSize) && !sort;pass++){
        sort=true;
        for(int x=0;x<listSize-pass;x++){
            if (data[x]>data[x+1]){
                tempValue=data[x];
                data[x]=data[x+1];
                data[x+1]=tempValue;
                sort=false;
            }
        }
    }

    for(int a=0;a<listSize;a++){
        cout<<data[a]<<" ";
    }
}

main(){
    int data[]={8,9,6,7,5,3,12};
    int sizearray=7;

    BubbleSort(data,sizearray);
}
```

d.

1	3	5	9	11	13	15	19
---	---	---	---	----	----	----	----

SELECTION SORTING (green = smallest value in the array, red = new position of the elements)

**NOTE: DESCENDING ORDER**

last	5	4	3	2	1
[0]	1	19	19	19	19
[1]	3	3	15	15	15
[2]	5	5	5	13	13
[3]	9	9	9	9	11
[4]	11	11	11	11	9
[5]	13	13	13	5	5
[6]	15	15	3	3	3
[7]	19	1	1	1	1

INSERTION SORTING

Pass	Comparison											Resultant array									
1	1	3	5	9	11	13	15	19				1	3	5	9	11	13	15	19		
2	1	3	5	9	11	13	15	19				1	3	5	9	11	13	15	19		
3	1	3	5	9	11	13	15	19				1	3	5	9	11	13	15	19		
3	1	3	5	9	11	13	15	19				1	3	5	9	11	13	15	19		
4	1	3	5	9	11	13	15	19				1	3	5	9	11	13	15	19		
5	1	3	5	9	11	13	15	19				1	3	5	9	11	13	15	19		
6	1	3	5	9	11	13	15	19				1	3	5	9	11	13	15	19		

Insertion sorting is significantly faster since its complexity ( $O(n)$ ) is lower compared to selection sorting ( $O(n^2)$ ).

## EXERCISE 2

a. Bubble Sort is based on compare and swap technique on every element in the array while Selection Sort repeatedly find the next largest (or smallest) elements in the array and place it in the correct place.

Question 2b (**note: DESCENDING ORDER**)

BUBBLE SORT

20	80	40	25	60	30
----	----	----	----	----	----

### Phase 1

80	20	40	25	60	30	Compare 20,80. Swap (20<80)
80	40	20	25	60	30	Compare 40,20. Swap (20<40)
80	40	25	20	60	30	Compare 20,25. Swap (20<25)
80	40	25	60	20	30	Compare 20,60. Swap (20<60)
80	40	25	60	30	20	Compare 20,30. Swap (20<30)
Total no of swap = 5, listSize = 5						

### Phase 2

80	40	60	25	30	20	Compare 25,60. Swap (25<60)
80	40	60	30	25	20	Compare 25,30. Swap (25<30)
Total no of swap = 2, listSize = 5						

### Phase 3

80	60	40	30	25	20	Compare 40,60. Swap (40<60)
Total no of swap = 1, listSize = 5						

SELECTION SORT (green = smallest value in the array, red = new position of the elements)

last	5	4	3	2	1
[0]	20	30	30	60	80
[1]	80	80	80	80	60
[2]	40	40	40	40	40
[3]	25	25	60	30	30
[4]	60	60	25	25	25
[5]	30	20	20	20	20

Question c

Pass	Comparison										
1	28	18	21	10	25	30	12	71	32	58	15
2	18	28	21	10	25	30	12	71	32	58	15
3	18	21	28	10	25	30	12	71	32	58	15
4	10	18	21	28	25	30	12	71	32	58	15
5	10	18	21	25	28	30	12	71	32	58	15

Pass	Resultant array										
1	18	28	21	10	25	30	12	71	32	58	15
2	18	21	28	10	25	30	12	71	32	58	15
3	10	18	21	28	25	30	12	71	32	58	15
4	10	18	21	25	28	30	12	71	32	58	15
5	10	18	21	25	28	30	12	71	32	58	15

### EXERCISE 3

**Note: code in descending order**

Question 3a (green = smallest value in the array, red = new position of the elements)

last	5	4	3	2	1
[5]	9	3	3	3	3
[4]	8	8	5	5	5
[3]	6	6	6	6	6
[2]	7	7	7	7	7
[1]	5	5	8	8	8
[0]	3	9	9	9	9

b. Selection sort

c. Descending order

d. In function `sort()`, the array list and the size of array are received as parameters. In the outer *for* loop, integer `last` is assigned to size of array subtracted by 1. The random integer `x` is assigned to 0. Moving on in the inner *for* loop, second element is compared to the first element. If the *if* statement is true, then latest value of `X` is assigned with value in integer `p`. The loop repeats until the last element has been compared. The final result shows that the largest value is taken to the first position in the array through descending order.

Question 3e (green = smallest value in the array, red = new position of the elements)

last	5	4	3	2	1
[5]	3	3	3	3	3
[4]	5	5	5	5	5
[3]	6	6	6	6	6
[2]	7	7	7	7	7
[1]	8	8	8	8	8
[0]	9	9	9	9	9

e. Array G has better efficiency of the algorithm compared to Array F. This is because elements in Array G does not need any sorting thus it is known as the best case for Selection sort.