Analysis of Algorithms
Project I
“Bubble Sort and Merge Sort Algorithms”

Student name: Medina Zaganjori
Student number: 150160908
Due date: 28.10.2019

Part a : Asymptotic lower bound AND upper bound.

In Bubble Sort algorithm we start from the 0th element and in the following loop we start from an adjacent element. In the inner loop body we compare each adjacent numbers and swap them if they are not in order. When one iteration finishes the “heaviest” element shows up at the end. So firstly we make n-1 comparisons and then n-2, n-3 and goes on.
Best case time and upper bound: $O(n^2)$.
Worst case and upper bound: O(n).

In Merge Sort Algorithm the list we want to sort will be divided in 2 arrays with equal size by separating the list on middle element. Each from the sub list will be sorted by using merge function (recursion) and then the sorted sub-lists are merged to form a completed list. By first dividing array into 2 equal halves log n will have a number of steps maximum log (n+1). The reason why the complexity for worst and best time is the same is because we are trying to divide the array in another arrays and then merges these last ones into one.

Merge		Merge Sort	
Best case time and upper bound:	O(n).	Best case time and upper bound:	O(n log n).
Worst case and upper bound:	O(n).	Worst case and upper bound:	O(n log n).

	Statement	Steps/ Execution	frequency		Total Steps	
			If-true	If-false	If-true	If-false
1	`void bubble(int *my_num,int &counter_run){`	1				
2	`for (int i = 0; i < counter_run; i++)`	1	N+1	N+1	N+1	N+1
3	{	0				
4	`for (int j = counter_run – 1; j >= i + 1; j--)`	1	N-2	N+1	N-2	N+1
5	{	0				
6	`if (my_num[j] < my_num[j – 1])`	1	N	1	N	1
7	{	0				
8	`int momentary = my_num[j];`	1	N	0	N	0
9	`my_num[j] = my_num[j - 1];`	1	N	0	N	0
10	`my_num[j - 1] = momentary;`	1	N	0	N	0
11	}}}	0				
					$O(n^2)$	O(n)

Figure1. Bubble Sort Algorithm

	Statement (Merge Sort function)	Steps/Execution	frequency		Total Steps	
			If-true	If-false	If-true	If-false
1	void merge_sort(int * my_num,int min,int max){	1	-	-	-	-
2	if(min<max){	1	1	1	1	1
3	int median = (max+min)/2;	1	1	1	1	1
4	merge_sort(my_num,min,median);	x	1	1	x	x
5	merge_sort(my_num,median+1,max);	x	1	1	x	x
6	merge(my_num,min,median,max);}}	O(n)	1	1	O(n)	O(n)
7	T(n) = 2 T(n/2) + O(n) Big O = O(nlogn)			2x + O(n)+2		

Figure2. Merge Sort Algorithm

	Statement (Merge function)	Steps/ Execution	frequency		Total Steps	
			If-true	If-false	If-true	If-false
1	void merge(int* my_num,int min,int median, int max){	-	-	-	-	-
2	int i=0; int j=0;	2	1	1	2	2
3	int size1,size2; // sizes of the momentary subarrays	2	1	1	2	2
4	size1=(median-min)+1;	1	1	1	1	1
5	size2=(max-median);	1	1	1	1	1
6	mom_max=new int[size1+1];	1	1	1	1	1
7	mom_min=new int[size2+1];	1	1	1	1	1
8	while(i< size1){	1	N/2+1	N/2+1	N/2+1	N/2+1
9	mom_min[i] = my_num[min+i];	1	N/2	N/2	N/2	N/2
10	i++;}	1	N/2	N/2	N/2	N/2
11	while(j< size2){	1	N/2+1	N/2+1	N/2+1	N/2+1
12	mom_max[j] = my_num[median+min+j];	1	N/2	N/2	N/2	N/2
13	j++;	1	N/2	N/2	N/2	N/2
14	}	-	-			
15	int k=0; int l=0; int m=min;	3	1	1	3	3
16	if(k< size1 && l< size2){	1	1	1	1	1
17	if(mom_min[k]<=mom_max[l]){	1	1	0	1	0
18	my_num[m]=mom_min[k];	1	1	0	1	0
19	k++;}	1	1	0	1	0
20	else{		-			
21	my_num[m]=mom_max[l];	1	0	1	0	1
22	j++;}	1	0	1	0	1
23	m++;}	1	0	1	0	1
24	while(l< size2){	1	N/2 +1	N/2 +1	N/2 +1	N/2 +1
25	my_num[m]=mom_min[l];	1	N/2	N/2	N/2	N/2
26	l++;	1	N/2	N/2	N/2	N/2
27	m++;	1	N/2	N/2	N/2	N/2
28	}	-	-	-		
29	delete mom_max;	1	1	1	1	1
30	delete mom_min;}	1	1	1	1	1
31					O(n)	Ω(n)

Figure3: Time Complexity of Merge Algorithm.

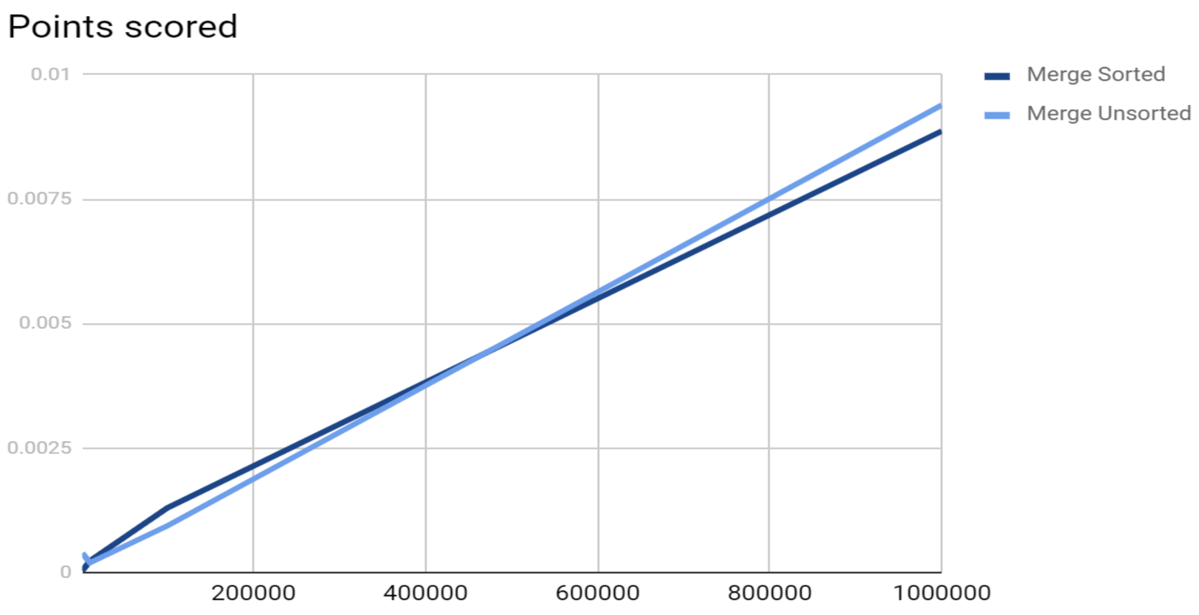
Part b: Average Time Execution of Algorithms

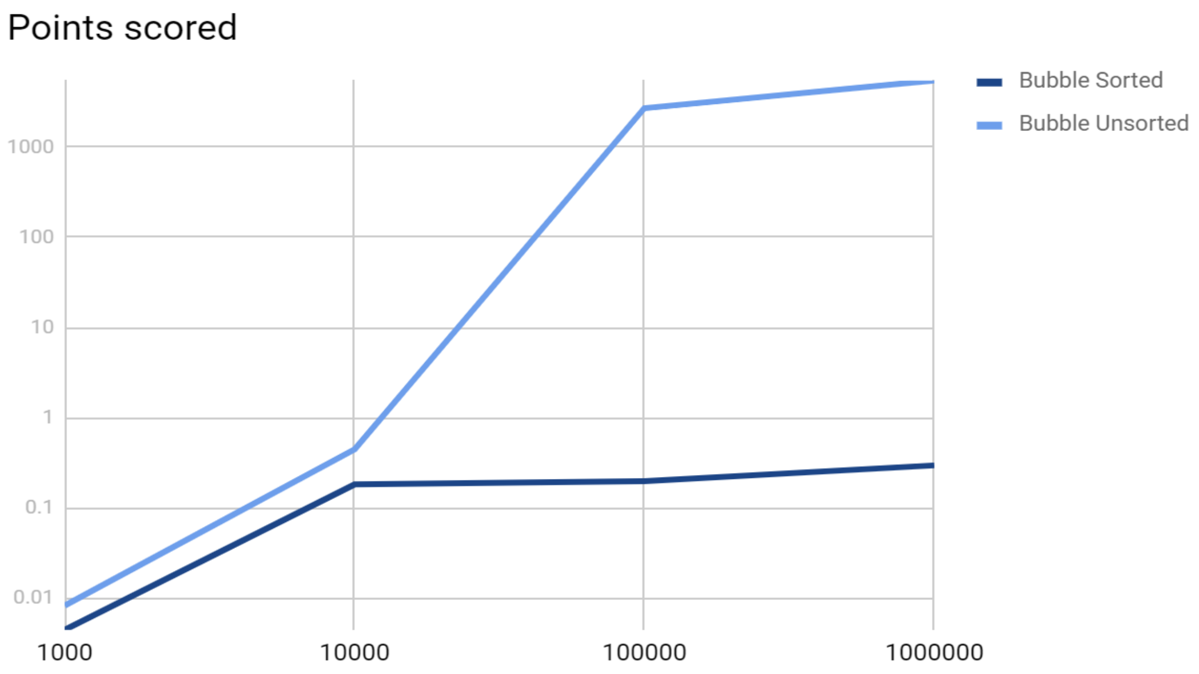
Bubble Sort Algorithm				
	1000	10000	100000	1000000
sorted.txt	0.004443s	0.18979s	0.001128s	0.008932s
unsorted.txt	0.008242s	0.445214s	47 mins	92 mins
Merge Sort Algorithm				
	1000	10000	100000	1000000
sorted.txt	0.000032s	0.000235s	0.001298s	0.008866s
unsorted.txt	3.9E-05s	0.000193s	0.000938s	0.000938s

Figure4. Time Average Table

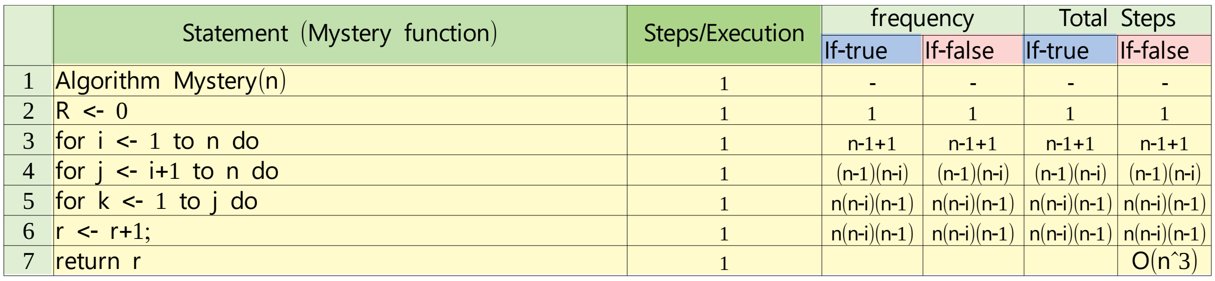
Part c: Run Time Complexities

For example, bubble sort has a complexity of O(n2). If you've got 10 elements, it would take about 100 steps to sort your list. But if you've got 50 elements, it would take about 2500 steps to sort. In contrast, a merge sort is O(n log n). With a 5 element list, it would take about 4 steps to sort it. With a 50 element list, it would take about 85 steps. In this part we were required to run our algorithms for different sizes of read data sets and by using clock function we wrote down the table below. Bubble Sort algorithm sorts very quickly when it comes to small data sets but when the size of data gets increased then the efficiency falls. Meanwhile Merge Sort becomes more efficient when the size of our data became 1000000. So as a conclusion bubble sort has O(n) in its best case and O(n 2) time complexity. At the same time Merge Sort takes O(n log(n)) time complexity.





Part d: Algorithm Mystery



	Statement (Mystery function)	Steps/Execution	frequency		Total Steps	
			If-true	If-false	If-true	If-false
1	Algorithm Mystery(n)	1	-	-	-	-
2	R <- 0	1	1	1	1	1
3	for i <- 1 to n do	1	n-1+1	n-1+1	n-1+1	n-1+1
4	for j <- i+1 to n do	1	(n-1)(n-i)	(n-1)(n-i)	(n-1)(n-i)	(n-1)(n-i)
5	for k <- 1 to j do	1	n(n-i)(n-1)	n(n-i)(n-1)	n(n-i)(n-1)	n(n-i)(n-1)
6	r <- r+1;	1	n(n-i)(n-1)	n(n-i)(n-1)	n(n-i)(n-1)	n(n-i)(n-1)
7	return r	1				O(nˆ3)

Figure9: Time Complexity of Algorithm

(n(n+1))/2 – (i(i+1))/2
Since “i” takes value from -2 to n. Then the expressions goes to n(n(n+1))/2-n(n+1)(2n+1)/12 – n(n+1)/4 so the order of our algorithm is O((n^3)/3). Since we have three different loops in each iteration of incrementing r value it considers a time step. Since this polynomial grows at the same rate n^3, then we could say that the function lies in the set Theta(n^3).