Analysis of Algorithms I
Assignment *I Report*

Part a) Finding asymptotic lower bounds and upper bounds for merge-sort and bubble-sort:

To find asymptotic bounds first we should implement Table for our Algorithms.After that we should find best-case for lower bounds because lowest time needed for an algorithm is the best-case scenerio and for this same reason we select worst-case for our upper bound.

Figure 1: Table for BubbleSort.

	Statement	Step/Execution	frequency		Total Steps	
			if-True	if-False	if-True	if-False
1	void AlgoSort::BubbleSort() {	0	-	-		
2	bool sorted = false;	1	1	1	1	1
3	int size = n;	1	1	1	1	1
4	while (size > 1 && sorted == false) {	2	n-1	n-1	2n-2	2n-2
5	int temp;	1	n-1	0	n-1	0
6	sorted = true;	1	n-1	0	n-1	0
7	for (int j = 1; j < size; j++) {	1	n+1	0	n+1	0
8	if (data[j] < data[j - 1]) {	1	n*(n-1)	0	n*(n-1)	0
9	temp = data[j];	1	n*(n-1)	0	n*(n-1)	0
10	data[j] = data[j - 1];	1	n*(n-1)	0	n*(n-1)	0
11	data[j - 1] = temp;	1	n*(n-1)	0	n*(n-1)	0
12	sorted = false;	1	n*(n-1)	0	n*(n-1)	0
13	}					
14	}					
15	size--;	1	n-1	0	n-1	0
16	}					
17	}					
					5n^2+n-2	2n
Total					O(n^2)	O(n)

From implementations in Figure 1,we see that best-case for our BubbleSort is O(n).Because if our data already sorted than we can say that our algorithm only executed for O(n) times.For worst-case scenerio our data must be in reverse order so we should do the same instructions over and over that time complexity will be equal to O(n^2).

Figure 2: Table For Merge() part of Recursive MergeSort.

	Statement	Step/Execution	frequency		Total Steps	
			if-True	if-False	if-True	if-False
1	void AlgoSort::Merge(int low, int mid, int high) {	0	-	-		
2	vector<int> temp(data);	1	1	1	1	1
3	int k = low, i = low, j = mid + 1;	3	1	1	3	3
4	while (i <= mid && j <= high) {	2	n/2+1	n/2+1	n+2	n+2
5	if (data[i] <= data[j]) {	1	n/2	n/2+1	n/2	n/2+1
6	temp[k] = data[i];	1	n/2	0	n/2	0
7	i++;	1	n/2	0	n/2	0
8	} else {					
9	temp[k] = data[j];	1	0	n/2	n/2	n/2
10	j++;	1	0	n/2	n/2	n/2
11	}					
12	k++;	1	n/2	n/2	n/2	n/2
13	}					
14	while (i <= mid) {	1	n/2+1	n/2+1	n/2+1	n/2+1
15	temp[k] = data[i];	1	n/2	0	n/2	0
16	k++;	1	n/2	0	n/2	0
17	i++;	1	n/2	0	n/2	0
18	}					
19	while (j <= high) {	1	n/2+1	n/2+1	n/2+1	n/2+1
20	temp[k] = data[j];	1	n/2	0	n/2	0
21	k++;	1	n/2	0	n/2	0
22	j++;	1	n/2	0	n/2	0
23	}					
24	for (int i = low; i < k; i++) {	1	n+1	n+1	n+1	n+1
25	data[i] = temp[i];	1	n	n	n	n
26	}					
27	}					
					10n+9	6n+10
Total					O(n)	O(n)

In Figure 2 we analysis Merge() function individually for best-case and worst-case. from basic steps we found best-case and worst-case as O(n).In our Recursive MergeSort implementation we use merge() function.

Figure 3: Table for Recursive MergeSort.

	Statement	Step/Execution	frequency		Total Steps	
			if-True	if-False	if-True	if-False
1	void AlgoSort::MergeSort(int low, int high) {	0	-	-		
2	if (low < high) {	1	1	1	1	1
3	int mid = (low + high) / 2;	1	1	1	1	1
4	MergeSort(low, mid);	T(n/2)	1	1	T(n/2)	T(n/2)
5	MergeSort(mid + 1, high);	T(n/2)	1	1	T(n/2)	T(n/2)
6	Merge(low, mid, high);	O(n)	1	1	O(n)	O(n)
7	}					
8	}					
					2T(n/2)+O(n)	2T(n/2)+O(n)
Total					n*log(n)	n*log(n)

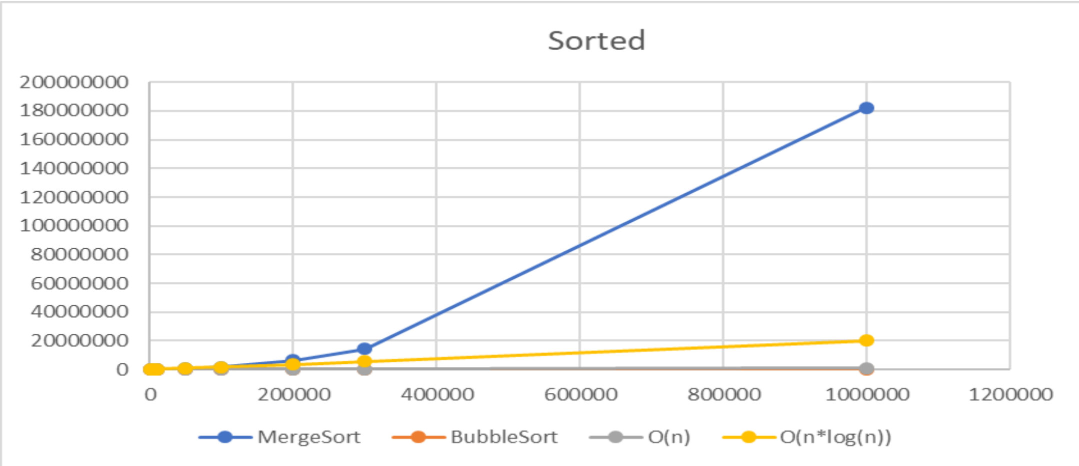
In this table we see that MergeSort calls itself so it is Recursive.And for merge() function part we already found best-case and worst-case for it so we can use same values for Figure 3 Merge() function.This Algorthim always divides itself to half so it is in form of log(n) and merge part is in form of O(n) because of this we choose n*log(n) for lower bound and upper bound.

Part b) Executing each search methods for each different value of N:

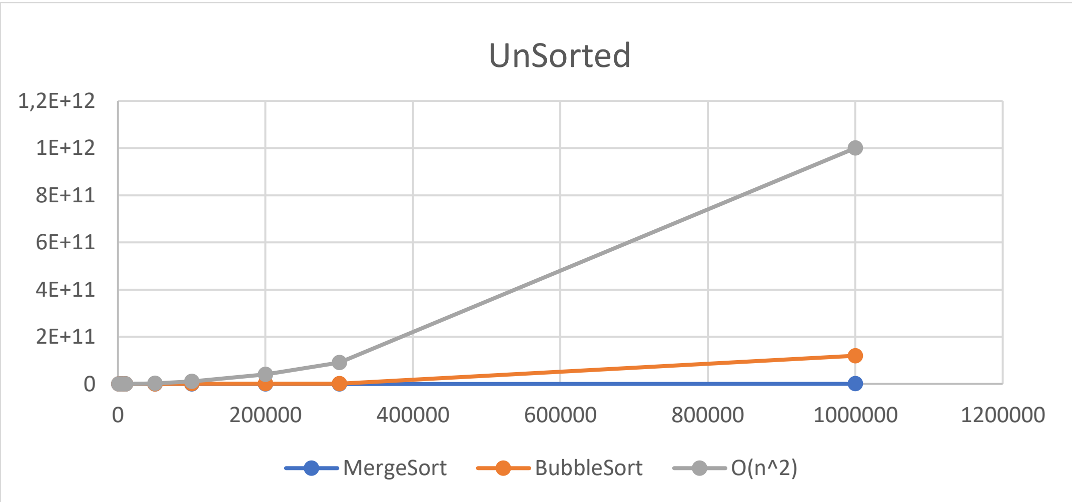
	1000	5000	10000	50000	100000	200000	300000
Merge-Sorted	716	6775	22052	414550	1620688	6300902	14171952
Merge-UnSorted	726	7132	23553	421795	1635584	6384462	14347304
Bubble-Sorted	10	51	99	518	1570	1992	3587
Bubble-UnSorted	10632	280524	1129722	28401916	113601358	456206094	1068424126

Figure 4 Each Search Method's execution time in microseconds for each different valuse for N.

Part c) Visualize BubbleSort and MergeSort and which cases you would choose which algorithm. Why?



While we work for Sorted Data we can see that BubbleSort is much efficient than our MergeSort algorithm so i would prefer BubbleSort for datas that are close to being sorted.



And in UnSorted Datas BubbleSort is Terrible and MergeSort is pretty reasonable so i would prefer MergeSort if data is completly random or very messy.

Part d) Mystery Function:

Statement	ep/Executi	frequency		Total Steps	
		if-True	if-False	if-True	if-False
Algorithm Mystery(n)	0	-	-		
r <- 0	1	1	1	1	1
for i <- 1 to n do	1	n+1	n+1	n+1	n+1
for j <- i+1 to n do	1	n*(n+1)	n*(n+1)	n*(n+1)	n*(n+1)
for k <- 1 to j do	1	n*n*(n+1)	n*n*(n+1)	n*n*(n+1)	n*n*(n+1)
r <- r+1;	1	n*n*(n+1)	n*n*(n+1)	n*n*(n+1)	n*n*(n+1)
return r	1	1	1	1	1
				2n^3+3n^2+2n+2	2n^3+3n^2+2n+2
Total				**O(n^3)**	**O(n^3)**

So it is O(n^3) Time Complexity;

Mystery function is f(n) = (n-2)^2 + (n-3)^2 + ...+ (1)^2.

And it's very poorly designed because it can be implemented easly as Time Complexity of O(n).