Problem 1

a) Pseudocode of Bubble Sort

BubbleSort (arr,n)	
for(i=1 to n)	
swap = false	//assume that swapping is not necessary
for(j=1 to n-1)	
if(arr[j] > arr[j+1])	//check if adjacent right-side element is //larger than the current element
swap(arr[j], arr[j+1])	//swap misplaced elements in the array
swap = true	//swapping happened
end if	
end for	
if(swap == false)	
break	//no more visit to the array by incrementing //the outer loop is required since all //elements are sorted
end for	
end BubbleSort	

The implementation of the algorithm and the test for time complexity measurement for bubble sort can be seen in BubbleSort.c.

b)

There are two nested arrays in the bubble sort. In the best case, the array will be always sorted, so the swap condition will remain false. In the first iteration of the outer loop, the inner will iterate for n-1 times, the swapping statement in the inner loop will always fail, executing the outer loop after one iteration. So, there will be (n-1) operations in the function.

Proof: Let $f(n) = n-1$, $g(n) = n$, $c=0.5$

$c.g(n) = 0.5n \geq f(n)$ if $n \geq 2$

Hence, the time complexity is $O(n)$.

In the worst case, all elements will be reversely sorted. The inner will work for n-i iterations for each outer iterations. Thus, the bubblesort function will work for $(n)(n-1)/2$ operations = $(n^2-n)/2$ operations.

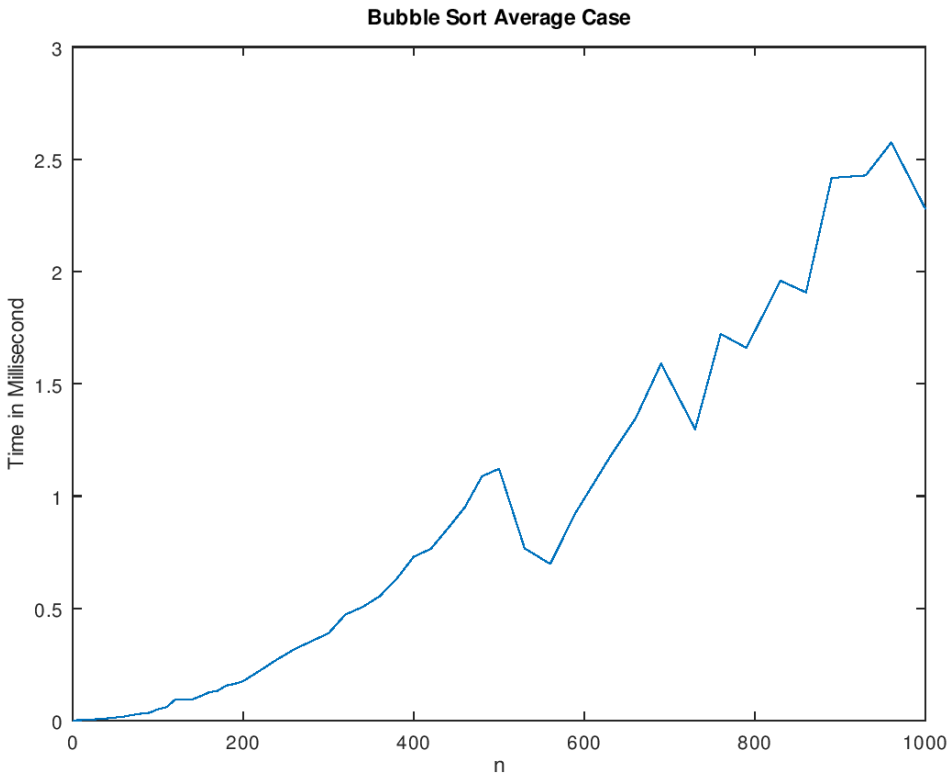
Proof: Let $f(n) = (n^2-n)/2$, $g(n) = n^2$, $c = 0.25$

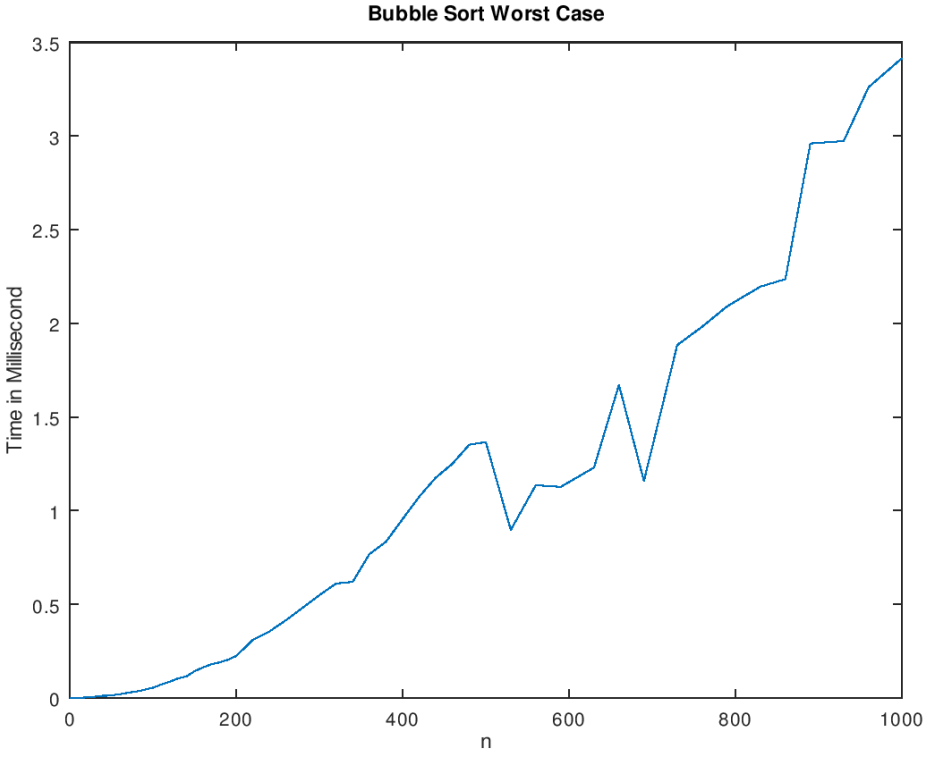
$cg(n) = 0.25n^2 \geq f(n)$ if $n \geq 2$

Hence, the time complexity is $O(n^2)$.

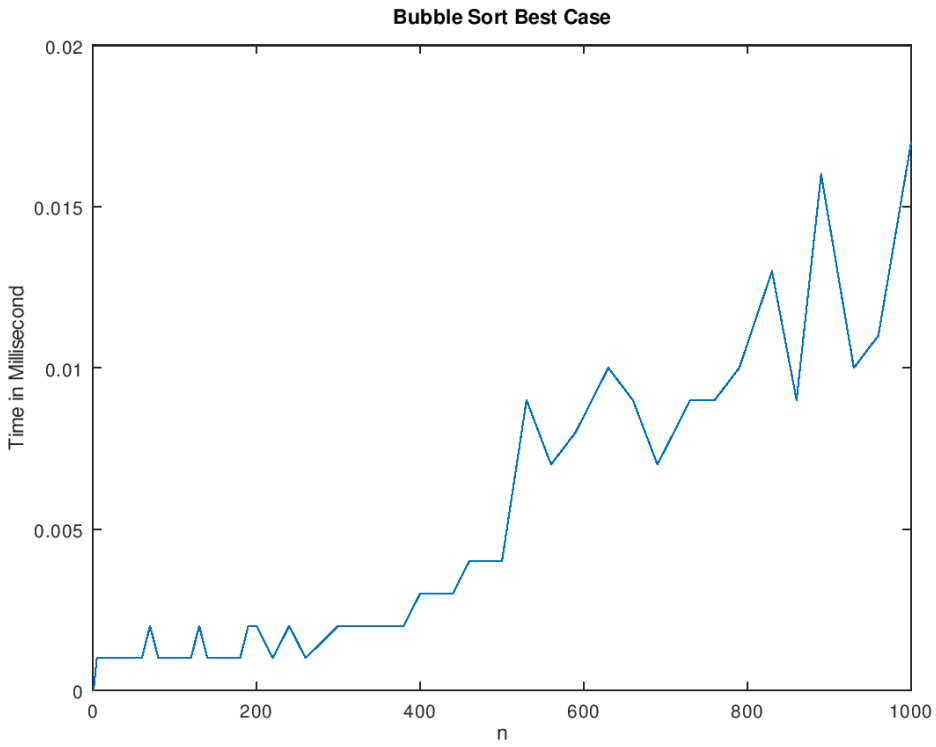
In the average case, we can say that the array is partially sorted, so we can divide the number of operations for the worst case by 2. Hence, the total operations would be $(n^2-n)/4$ operations = $(n^2/4) – (n/4)$ operations

Proof: Let $f(n) = (n^2-n)/4$, $g(n) = n^2$, $cg(n) \geq f(n)$ if $c = 1>0$. Hence, the time complexity is $O(n^2)$.

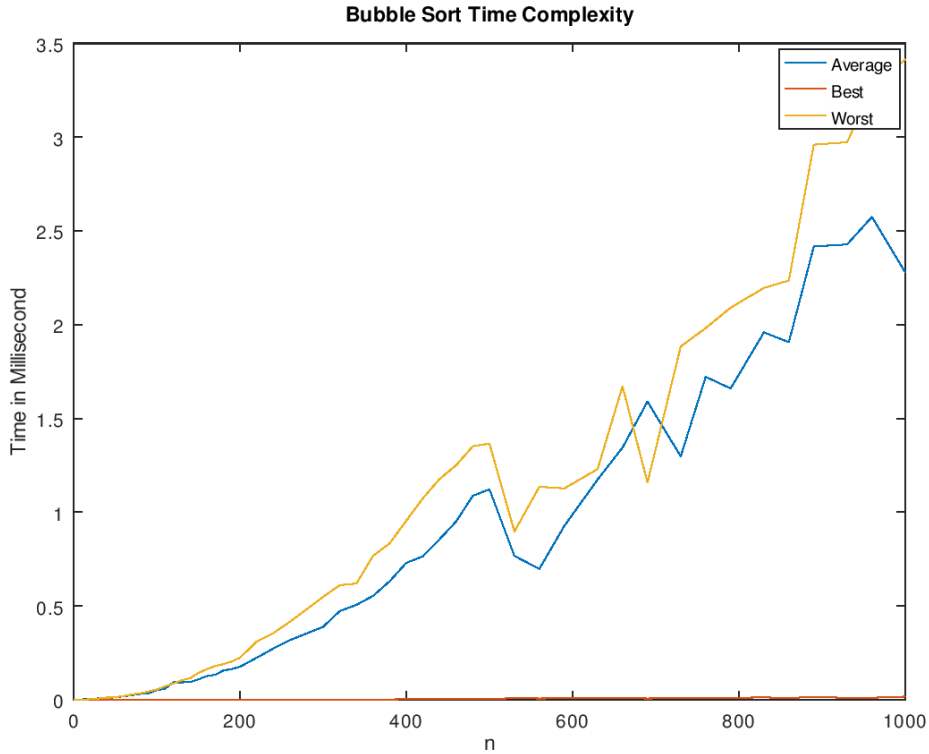




Time taken growth rate of average case and worst case graphs tend to look like quadratic function graphs.



The time taken growth rate of best case looks like a linear function graph.



Problem 4.1

c) Insertion Sort: Stable

Once the element's position is confirmed (while sorting to be in the correct order), that element's position will not be changed anymore. If the value of R and S are equal and R appeared before S in the original array, R will also appear before S in the sorted array as the position of R will set first, and insertion sort swaps the elements if and only if the current element is larger than the adjacent element on the right side, so it does not affect equal elements. Also, the smallest sorted elements will always stay stable in their sorted positions on the left side as the algorithm will ignore the sorted left-side elements after each iteration.

Merge Sort: Stable

If the merge operation is implemented properly, the order of same key elements will be remained as being split, so the order cannot be changed. If R and S are in the middle of the array to be sorted, R will be in the left split array and S in the right split array. While merging, R will still be sorted before S as the merging statement checks whether the pointed element in the left-side array is less than or equal to the element in the same index of the right-side array. (R <= S). If the statement is true, R will be allocated first, and then S.

Heap Sort: Unstable

The positions and order of all elements become lost once heap sort algorithm has started. If R is connected under nodes which are left-sided after the root and S under the nodes on the right side of the root, there is a high probability that R will be disconnected from the node first while rearranging the order, and then S. In this way, S will appear in front of R. So, heap sort is unstable.

Bubble Sort: Stable

The swapping statement for bubble sort works if and only if the current element is larger than the right-side adjacent element. If R and S are equal and R appears before S in the original array, R will still appear before S as the algorithm does not consider re-ordering or swapping same key elements.

Also, the sorted largest elements will stay stably on the right side of the array as their positions will be ignored after each iteration of the outer loop until the whole array is sorted successfully.

d) Insertion Sort: Adaptive

The time complexity for the best case is O(n). The time complexity for average case and worst case are $O(n^2)$. So, if the original array is nearly sorted, there will be less swapping operations required to place smallest elements on the left side of the array, and the time

taken for insertion sort will be nearly linear. So, the sorting process will be finished faster.

Merge Sort: Non-Adaptive

The time complexity for the best case, worst case and average case are O(n log (n)). It means that the number of merge sort operations does not depend on the order of elements in the original array. If there are two array with same number of elements, one in nearly sorted mode, and one almost near to the worst case, the time taken for both arrays will be nearly equal as all elements in both arrays will be split and then merged regardless of their orders.

Heap Sort: Non-Adaptive

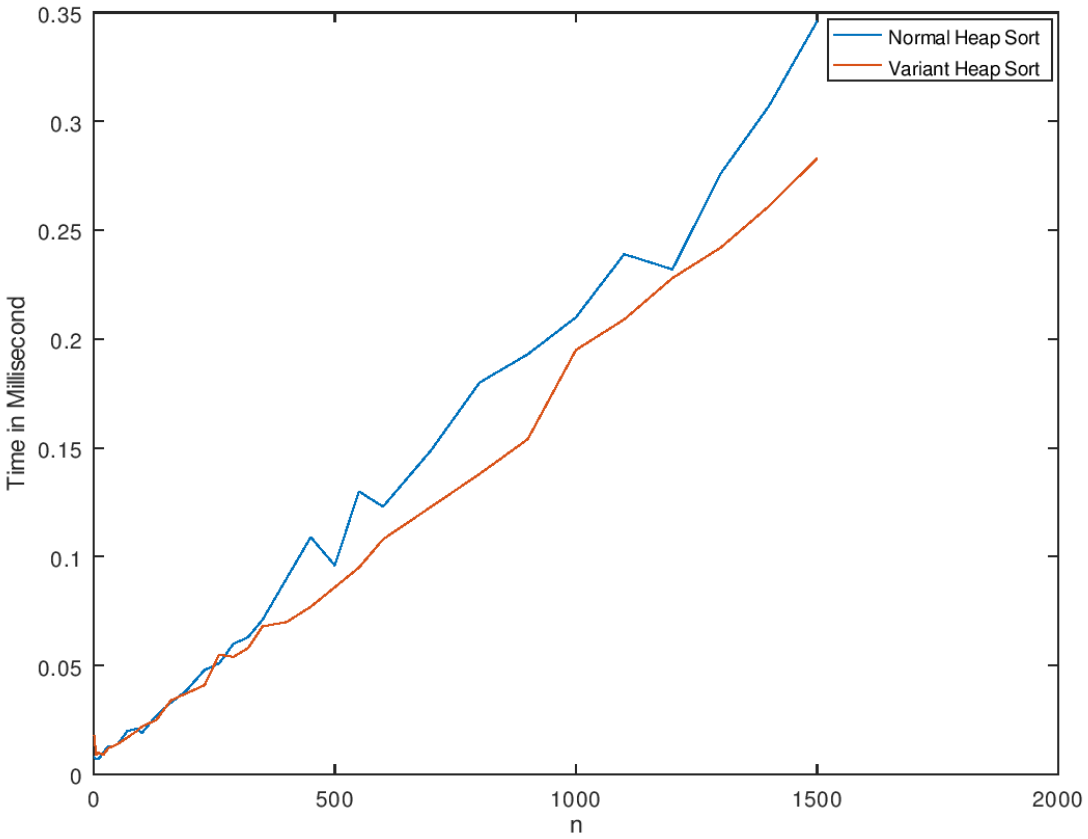
The time complexity for the best case, worst case and average case are O(n log (n)). So, the number of merge sort operations does not depend on the order of elements in the original array. If there are two array with same number of elements, one in nearly sorted mode, and one almost near to the worst case, the time taken for both arrays will be nearly equal as all elements will be heapified, compared, and disconnected from nodes to sort, regardless of their orders.

Bubble Sort: Adaptive

The time complexity for the best case is O(n). The time complexity for average case and worst case are $O(n^2)$. So, if the original array is nearly sorted, there will be less swapping operations to place largest number on the right side of the array, and the time taken for bubble sort will be nearly linear. So, the sorting process will be finished faster.

Problem 4.2

a) The implementation of HeapSort is in HeapSort.c file.
b) The implementation of Variant HeapSort is Variant-Heap.c file.
c) The figure below indicates the average time taken comparison between normal heap sort algorithm and variant heap sort algorithm. The time taken for the same n is measured for at least 5 times for each method by using the same files for a) and b), and the average time taken measurements are added in heapsort.txt and bottomup.txt. Afterwards, the time taken for each function is read and plotted by using Matlab. According the measurements using two files, the variant heap sort takes less significant time than the normal heap sort algorithm if n becomes larger than 400. In the lecture, it is mentioned that the time complexity for normal heap sort average case is O(n log(n)). Since the variant heap sort takes less time than the normal one, its time complexity is assumed to be O(n).


Normal Heap Sort
Variant Heap Sort
Time in Millisecond
n
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0
500
1000
1500
2000