a) c = 8 & n0 = 1 ; 7 < 8
Explanation:

$6n^4 + 9n^2 - 8 \leq cn^4$

$6 + 9/n^2 - 8/n^4 \leq c$

$1/n^4 = 1, n = 1$

7 <= c

b) Trace the below mentioned sorting algorithms to sort the array [ 5, 3, 2, 6, 4, 1, 3, 7 ] in ascending order.

i) **Selection Sort :** Find the biggest element in the array and swap it with the last index.
Red = last index tracker, Highlighted = Biggest element until the tracker
[ 5, 3, 2, 6, 4, 1, 3, 7 ]
5, 3, 2, 6, 4, 1, 3, 7
5, 3, 2, 6, 4, 1, 3, 7
5, 3, 2, 3, 4, 1, 6, 7
1, 3, 2, 3, 4, 5, 6, 7
1, 3, 2, 3, 4, 5, 6, 7
1, 3, 2, 3, 4, 5, 6, 7
1, 2, 3, 3, 4, 5, 6, 7
1, 2, 3, 3, 4, 5, 6, 7

In selection sort algorithm, there are 2 for loops that execute the array n-1 times.
In this example case, one for loop executes 7 times.
Number of key comparisons: n*(n-1)/2. In this example case, it does 28 comparisons.
Since it does not depend on the initial organization of data; best, worst and average case are the same and they are O(n^2).

ii) **Merge Sort:** Merge sort is a divide & conquer algorithm and it uses recursion. Firstly, it divides the given array into two half and sorts each part separately, then it merges into one sorted array.
Red = left handside, green = right handside
[ 5, 3, 2, 6, 4, 1, 3, 7 ]
5, 3, 2, 6 4, 1, 3, 7
5, 3
5 3 -> base cases (return)
3, 5 -> merged
2, 6
2 6 -> base cases (return)
2, 6 -> merged
2, 3, 5, 6 -> merged
4, 1
4 1 -> base cases (return)
1, 4 -> merged

3, 7
3 7 -> base cases (return)
3, 7 -> merged
1, 3, 4, 7 -> merged
1, 2, 3, 3, 4, 5, 6, 7 -> merged
merge() method does n-1 comparisons.
T(n) = T(n/2) + n-1

iii) **Quick Sort – Assume the last element is chosen as a pivot**.
[ 5, 3, 2, 6, 4, 1, 3, 7 ]
Red = J index Green = i index
Partition for pivot = 7:
[ 5, 3, 2, 6, 4, 1, 3 , 7]
[ 5, 3, 2, 6, 4, 1, 3 , 7]
[ 5, 3, 2, 6, 4, 1, 3 , 7]
[ 5, 3, 2, 6, 4, 1, 3 , 7]
[ 5, 3, 2, 6, 4, 1, 3 , 7]
[ 5, 3, 2, 6, 4, 1, 3 , 7]
[ 5, 3, 2, 6, 4, 1, 3 , 7]
[ 5, 3, 2, 6, 4, 1, 3 , 7] -> swap(arr[right], arr[i]); return 7 as new pivot index
Left array = [ 5, 3, 2, 6, 4, 1, 3]
Right array = []
For the left array with pivot = 3:
[ 5, 3, 2, 6, 4, 1, 3]
[ 5, 3, 2, 6, 4, 1, 3] -> i--; swap(arr[j], arr[i]); //arr[j] > pivot
[ 5, 3, 2, 6, 1, 4, 3] -> i--; swap(arr[j], arr[i]); //arr[j] > pivot
[ 5, 3, 2, 1, 6, 4, 3]
[ 5, 3, 2, 1, 6, 4, 3]
[ 5, 3, 2, 1, 6, 4, 3] -> i--; swap(arr[j], arr[i]); //arr[j] > pivot
[1, 3, 2, 5, 6, 4, 3] -> swap(arr[right], arr[i]);
[1, 3, 2, 3, 6, 4, 5] -> return 3 as pivot index
Left array = [ 1, 3, 2]
Right array = [6, 4, 5]
For the left array with pivot = 2:
[ 1, 3, 2] -> i--; swap(arr[j], arr[i]); //arr[j] > pivot
[ 1, 3, 2]
[ 1, 3, 2] -> swap(arr[right], arr[i]);
[ 1, 2, 3] -> return 1 as new pivot index
Left array = [ 1] ->return
Right array = [3] ->return
For the right array with pivot = 5:
[6, 4, 5]
[6, 4, 5] -> i--; swap(arr[j], arr[i]); //arr[j] > pivot
[4, 6, 5] -> swap(arr[right], arr[i]);
[4, 5, 6] -> return 5 as pivot index
Left array = [4] ->return
Right array = [6] ->return

**c) Find the asymptotic running times in big O notation of $T(n) = 2T(n-1) + n^2$ , where by using the $T(1) = 1$ repeated substitution method. Show your steps in detail.**

T(1) = 1
n=2; T(2) = 2T(1) + 4 =2*1 + 4
n=3; T(3) = 2T(2) + 9 = 2(2T(1)) + 9
n=4; T(4) = 2T(3) + 16 =2(2(2T(1)) + 9) + 16
n=5; T(5) = 2T(4) + 25 =2(2(2(2T(1)) + 9) + 16) + 25

…

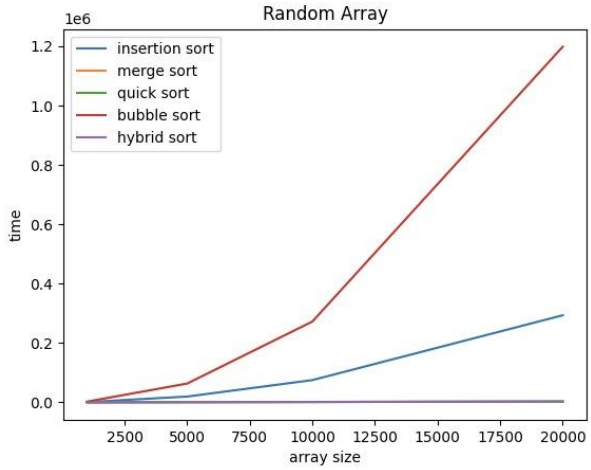
---

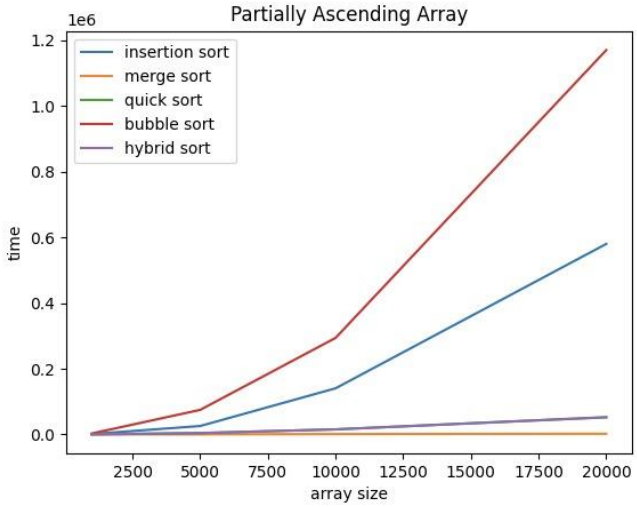
*** General expression
n=k; T(k) = 2T(k-1) + k*k =2 ( 2T(k-2) +(k-1)^2) + k^2
= 2^(k-1) * T(1) + $\sum_{i=0}^{k} 2^{k-i}(i)^2$

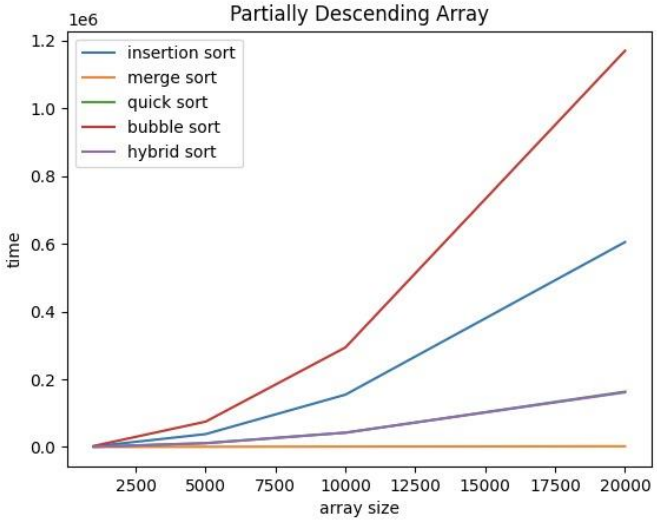
T(1) = 1 ; The result is 2^(n-1) + $\sum_{i=0}^{n} 2^{n-i}(i)^2$

T(n) ≅ 2^(n-1) 2^n(n+1)(2n+1)/6

| Array | Elapsed Time (ms) | | | | | Number of Comparisons | | | | | Number of Data Moves | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Insertion Sort | Bubble Sort | Merge Sort | Quick Sort | Hybrid Sort | Insertion Sort | Bubble Sort | Merge Sort | Quick Sort | Hybrid Sort | Insertion Sort | Bubble Sort | Merge Sort | Quick Sort | Hybrid Sort |
| R1K | **0.789** | **2.238** | **0.124** | **0.129** | **0.102** | **249277** | **499500** | **8682** | **10683** | **10683** | **250276** | **249277** | 9976 | **5863** | **5863** |
| R5K | **19.896** | **66.034** | **0.82** | **0.62** | **0.71** | **6282177** | **12497500** | **55208** | **65888** | **65888** | **6287176** | **6282177** | **61808** | **35190** | **35190** |
| R10K | **77.09** | **289.864** | **1.64** | **1.37** | **1.354** | **24843546** | **49995000** | **120463** | **149814** | **149814** | **24853545** | **24843546** | **133616** | **79349** | **79349** |
| R20K | **305.3** | **1216.19** | **3.28** | **2.98** | **2.93** | **99611258** | **199990000** | **260808** | **322291** | **322291** | **99631257** | **100153536** | **287232** | **182490** | **182490** |
| A1K | **0.025** | **1.13** | **0.08** | **0.3** | **0.28** | **2452** | **499500** | **5838** | **135736** | **135736** | **3451** | **2452** | **9976** | **1896** | **1896** |
| A5K | **0.135** | **26.747** | **0.49** | **0.42** | **5.382** | **16501** | **12497500** | **35900** | **2776781** | **2776781** | **21500** | **16501** | **61808** | **10594** | **10594** |
| A10K | **0.27** | **106.614** | **1.099** | **20.224** | **20.527** | **34998** | **49995000** | **77148** | **10490007** | **10490007** | **44997** | **34998** | **133616** | **21613** | **21613** |
| A20K | **0.536** | **416.569** | **2.12** | **75.402** | **76.610** | **74934** | **199990000** | **165098** | **39850863** | **39850863** | **94933** | **74934** | **287232** | **44886** | **44886** |
| D1K | **1.523** | **2.919** | **0.069** | **2.333** | **2.288** | **493566** | **499500** | **5890** | **481773** | **481773** | **494565** | **493566** | **9976** | **243105** | **243105** |
| D5K | **37.808** | **73.69** | **3.427** | **21.874** | **26.719** | **60.473** | **12497473** | **12497500** | **29810** | **12437634** | **12437634** | **12497473** | **61808** | **6225056** | **6225056** |
| D10K | **1504.456** | **2970.525** | **20.88** | **242.57** | **237.783** | **49915120** | **49995000** | **74534** | **49915128** | **49915128** | **49925119** | **49915120** | **133616** | **25009991** | **25009991** |
| D20K | **22431.531** | **47104.783** | **88.94** | **1364.66** | **1367.27** | **199830112** | **199990000** | **159136** | **199830120** | **199830120** | **199850111** | **199830112** | **287232** | **100019991** | **100019991** |


Random Array
1e6
insertion sort
merge sort
quick sort
bubble sort
hybrid sort
time
array size
0.0
0.2
0.4
0.6
0.8
1.0
1.2
2500
5000
7500
10000
12500
15000
17500
20000


Partially Ascending Array
1e6
insertion sort
merge sort
quick sort
bubble sort
hybrid sort
time
array size
0.0
0.2
0.4
0.6
0.8
1.0
1.2
2500
5000
7500
10000
12500
15000
17500
20000


Partially Descending Array
1e6
insertion sort
merge sort
quick sort
bubble sort
hybrid sort
time
array size
0.0
0.2
0.4
0.6
0.8
1.0
1.2
2500
5000
7500
10000
12500
15000
17500
20000

**General Analyses of The Graph and The Table**

First of all, almost every step hybrid sort's complexity is very closer and almost equal to quick sort's complexity because **we deal with arrays larger than 20 elements and that is why hybrid sort calls quick sort every time if we would call it with less than 20 elements, it would behave like bubble sort.**

With random array cases, we should consider algorithms' average time complexities because every element has equal chance to be any location and, in this case, the time spending is like the following: bubble sort > insertion sort > merge sort > quick sort = hybrid sort. Bubble sort's complexity is closer to insertion sort and merge sort's complexity is closer to quick sort but there is a sharp gap between these two groups because first group's average time complexity is O(n^2) and the second group's average time complexity is O(nlogn).

With partially ascending array cases, we should expect the results to be closer to algorithms' best time complexities because small partials of the arrays are already sorted. I mean for example, it is certain that the first log(2 to n) elements are smaller than the second log(2 to n) elements, in this case, the time spending is like the following: bubble sort > insertion sort > merge sort > quick sort = hybrid sort. Bubble sort's complexity is closer to insertion sort and merge sort's complexity is closer to quick sort but there is a sharp gap between these two groups because first group's average time complexity is O(n^2) and the second group's average time complexity is O(nlogn).

With partially descending array cases, we should expect the results to be closer to algorithms' worst time complexities because small partials of the arrays are already sorted descendingly. I mean for example, it is certain that the first log(2 to n) elements are greater than the second log(2 to n) elements, in this case, the time spending is like the following: bubble sort > insertion sort > quick sort = hybrid sort > merge sort . Bubble sort's complexity (O(n^2)) is closer to insertion sort (O(n^2)). Merge sort's complexity (O(nlogn)) is lower than quick sort(O(n^2)) and with larger element such as 20000, the gap between merge sort's outputs and quick sort's outputs, can be seen clearly.

**Part 2-b-1 Analyses**

In the Part 2-b-1, we create the arrays with the RandomArrayGenerator call. The arrays contain unique numbers from 1 to n in random order. Thus, we expect that the elapsed times should be closer to the algorithms' average time's duration with n = array size. The average cases of insertion sort is O(n^2), of bubble sort is O(n^2), of merge sort is O(nlogn), of quick sort is O(nlogn) and of hybrid sort is O(nlogn). The reasons are the following:

In insertion sort, the number of swaps is equal to number of insertions. So, it is a comparison based algorithm and the input array's ordering is the actual thing that matters to determine its time complexity. If we assume that the array contains unique elements, the specific element that we compare can be greater or lower than the compared items. So, the probability of being greater is ½. Since there will be n(n-1)/2 comparisons in the algorithm because of the loops. The possibility of every item to be inverted is n(n-1)/2 * ½. Thus, there will be O(n^2) inversions and therefore, the expectation runtime will be O(n^2). From this explanation we can also understand that the average comparison number in the insertion sort is n(n-1)/4 because every item in a random ordered array is equally likely to be any position. In addition, since we say that swaps are equal to the insertion number. The average move number of insertion sort is n(n-1)/4. (n*(n-1)/4 is approximately equals to (n^2)/4 .)

The average time complexity of bubble sort is O(n^2) because in its worst case it compares and swaps every adjacent element which would do (n-1)^2 comparisons and swaps. On the other hand, in its

best case, it does not do (n-1)^2 swaps but it still does (n-1)^2 comparisons. Therefore the average case's complexity is also (n-1)^2 = O(n^2). In the average case, we assume that only the half of the array is sorted. For this reason, there will be (n-1)^2/2 * ½ swaps, approximately n^2 /4. Since we compare all item pairs in the array the average comparison count is (n-1)^2 / 2, approximately n^2 /2.

The average case of the merge sort is O(nlogn) because it uses divide and conquer method. In each recursive call of mergeSort(), the array is divided by 2 and then, two part merge. Dividing the array into two subarrays tooks O(1) time but merge part tooks O(n) time because it passes through the arrays at the same time. Since O(n) part is done log(2 to n) time. Overall complexity is O(nlogn) which does not depend on the array is sorted or not.

The average case of the quick sort is O(nlogn). Partition method which determines the new pivot index, takes O(n) times and after this method it does two recursive calls to divide the array into to half(start to pivot and pivot+1 to end) that takes O(logn) time. In total, this algorithm takes O(nlogn) time in average case. It is important that the number of comparisons and moves, depends on the pivot choice and the input array. In my case, I chose last element as the pivot point but if I chose some point near the middle (but not the exact middle point), my comparison and move numbers could increase.

**Part 2-b-2 & Part 2-b-3 Analyses**

Insertion sort algorithm is a comparison-based algorithm. Therefore, the time complexity of the best case and the worst case is the same with its average case because the comparison numbers are the same which is equal to (n-1)^2 comparisons and overall time complexity is O(n^2).

Bubble sort algorithm is a comparison-based algorithm. Therefore, the time complexity of the best case and the worst case is the same with its average case because the comparison numbers are the same which is equal to (n-1)^2 comparisons and overall time complexity is O(n^2).

Merge sort is a divide and conquer based algorithm. Therefore, every time it does two recursive call and an O(n) time method call. So, its overall time complexity is O(n^2) independent from the input array.

Quick sort's best case is the same with its average case because it still does n-1 comparisons in partition part and then calls recursive methods. Therefore, it takes O(nlogn) time. On the other hand, descending ordered array is one of its worst cases and its time complexity is different than its average case. Since it is partially ordered descending, each partition should result with 2 subarrays; one subarray ranging in size from 1 to [log(2 to n)-1] and one subarray with size of [n - (first subarray element number)]. Thus, there will be n*(n-1) / 2 operations and time complexity will be O(n^2) in the worst case.

**Results**

The values that I found are generally closer to my expectations. There are some slight deviations which my computer may cause to. For example, I would expect to find the same results with quick sort and hybrid sort but generally hybrid takes slightly less time than quick sort in my computer. Or I would expect to find closer results with bubble and insertion sort because in theory both takes O(n^2) time, but bubble sort takes more time in my experiment. To conclude, I see that array size and array's element distribution affects the time complexity. With random distribution but in small numbers, there are just small differences, and I may use insertion, bubble, hybrid, quick and merge sort but in large numbers it is best to use quick, hybrid or merge sort. With ascending distribution, I may consider same cases with random distribution, but the algorithms' execution takes less time. With descending distribution, the algorithms' execution takes more time and I may prefer to use merge sort instead of quick and hybrid sort when I deal with larger arrays.