Chapter 14 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. If you want to sort an array of n elements, which of the following statements is **FALSE**?
   A) The worst-case time complexity of mergesort is $\Theta(n\log(n))$
   B) The worst-case time complexity of quicksort is $\Theta(n\log(n))$
   C) The worst-case time complexity of heapsort is $\Theta(n\log(n))$
   D) The worst-case time complexity of bubble sort is $\Theta(n^2)$
   E) The worst-case time complexity of insertion sort is $\Theta(n^2)$

2. You decided to run the same sorting algorithm on two different arrays of size n, one of which is nearly sorted, while the other is not close to being sorted at all. After running the sorting algorithm, you notice that it took less time to sort the nearly-sorted vector. Which of the following is most likely to be the sorting algorithm that you used?
   A) Insertion sort
   B) Selection sort
   C) Heapsort
   D) Mergesort
   E) Quicksort

3. The university registrar has developed a new method for selecting people off the waitlist. Students with more credits are given priority over students with fewer credits. Ties in credit hours are settled with arrival order — that is, students who signed up for the waitlist earlier are selected over students who signed up later. If you could only look at the number of credits, which of the following sorts would **NOT** allow the registrar to accomplish these goals?
   A) Bubble sort
   B) Selection sort
   C) Insertion sort
   D) Mergesort
   E) All of the above sorts would satisfy the registrar's requirements

4. Which of the following statements is **TRUE**?
   A) Bubble sort can be implemented to be either adaptive or stable, but not both
   B) Insertion sort is fast on nearly sorted arrays but requires $\Theta(n)$ auxiliary space
   C) Selection sort performs only $\Theta(n)$ comparisons on an already sorted array
   D) The best-case time complexity of selection sort is $\Theta(n^2)$
   E) None of the above

5. Two of your EECS 281 friends, Daniel and Ryan, want your help with selecting a sorting algorithm that best suits their needs. Daniel has an array that holds identical keys, and he does not want the sorting algorithm to modify the order of these identical elements. However, he does not care about the auxiliary space that his sorting algorithm may require. Ryan, on the other hand, desires the opposite: he wants a sorting algorithm that can guarantee him $\Theta(1)$ auxiliary space in the worst case for sorting his array, but he does not mind if the sort randomizes the order of identical keys in the process. Both students want a sorting algorithm with a worst-case time complexity of $\Theta(n\log(n))$. Given these requirements, which of the following is an acceptable recommendation for both Daniel and Ryan?
   A) Daniel should use heapsort, and Ryan should use quicksort
   B) Daniel should use quicksort, and Ryan should use heapsort
   C) Daniel should use heapsort, and Ryan should use mergesort
   D) Daniel should use mergesort, and Ryan should use heapsort
   E) More than one of the above

6. One way to classify sorting algorithms is by their adaptability. What is an advantage of a non-adaptive sorting algorithm?
   A) Non-adaptive sorting algorithms may perform less work depending on the order of the data
   B) Non-adaptive sorting algorithms are always faster than adaptive algorithms
   C) Non-adaptive sorting algorithms may change its sequence of operations based on the input
   D) Non-adaptive sorting algorithms may be simpler to implement than adaptive algorithms
   E) Non-adaptive sorting algorithms are always guaranteed to be stable

7. If you want to sort a collection of values that is too large to fit in the main memory of a computing device, one strategy is to use an *external sorting* algorithm that divides the input into smaller chunks that are small enough to fit in main memory, sorts them individually, and recombines the sorted chunks to obtain the full sorted output. This external sort is most similar to which of the following sorting algorithms?
   A) Bubble sort
   B) Selection sort
   C) Insertion sort
   D) Heapsort
   E) Mergesort

8. Which of the following sorting algorithms can be used to sort a linked list of size n in $\Theta(n\log(n))$ time, while ensuring that identical elements remain in the same relative order before and after the sort?
   A) Selection sort
   B) Heapsort
   C) Mergesort
   D) Quicksort
   E) None of the above

9. Which of the following statements is/are **TRUE**?
   I. When sorting an array of size n, mergesort requires $\Theta(n)$ auxiliary space.
   II. The fastest possible comparison sort has a worst-case time complexity no better than $\Theta(n\log(n))$.
   III. Quicksort will always sort an array faster than bubble sort.

   A) I only
   B) II only
   C) I and II only
   D) I and III only
   E) I, II, and III

10. You are using quicksort to sort the following 9 elements in ascending order:

```
int32_t arr[] = {43, 82, 50, 65, 24, 19, 78, 16, 37}
```

    Which of the following elements would be the **best** choice for the pivot?
    A) 43
    B) 50
    C) 24
    D) 37
    E) None of the above

11. Your friend is attempting to implement top-down mergesort, but they incorrectly implemented their merging algorithm to take $\Theta(n^2)$ time instead of $\Theta(n)$ time, given an input of size n. Assuming that all other steps of the mergesort algorithm are implemented correctly, what is the time complexity of your friend's mergesort algorithm?
    A) $\Theta(n\log(n))$
    B) $\Theta(n^2)$
    C) $\Theta(n^2\log(n))$
    D) $\Theta(n^3)$
    E) $\Theta(n^3\log(n))$

12. Consider two implementations of quicksort. Quicksort A always chooses the rightmost (last) element within the subarray as the pivot, while quicksort B always chooses the smallest value within the subarray as the pivot. For example, given the subarray [5, 3, 1, 2, 4], quicksort A would choose 4 as the pivot, while quicksort B would choose 1 as the pivot. If both algorithms can find the correct pivot in *constant time* with each partition, which of the following statements are **TRUE**?
    A) The average-case time complexity of quicksort A is *better* than the average-case time complexity of quicksort B.
    The worst-case time complexity of quicksort A is *better* than the worst-case time complexity of quicksort B.
    B) The average-case time complexity of quicksort A is *equal* to the average-case time complexity of quicksort B.
    The worst-case time complexity of quicksort A is *equal* to the worst-case time complexity of quicksort B.
    C) The average-case time complexity of quicksort A is *worse* than the average-case time complexity of quicksort B.
    The worst-case time complexity of quicksort A is *worse* than the worst-case time complexity of quicksort B.
    D) The average-case time complexity of quicksort A is *worse* than the average-case time complexity of quicksort B.
    The worst-case time complexity of quicksort A is *equal* to the worst-case time complexity of quicksort B.
    E) The average-case time complexity of quicksort A is *better* than the average-case time complexity of quicksort B.
    The worst-case time complexity of quicksort A is *equal* to the worst-case time complexity of quicksort B.

13. You are given three mystery sorting algorithms, sorts A, B, and C. To determine their identities, you run these sorts multiple times on three different arrays, each containing 100000 integer values. The results you obtain are shown in the table below:

	Sort A	**Sort B**	**Sort C**
average runtime on unsorted array (seconds)	6.62	0.00759	7.69
average runtime on nearly sorted array (seconds)	6.41	3.88	0.000211
average runtime on sorted array (seconds)	6.13	4.27	0.000073

    Knowing this information, which one of the following conclusions is most consistent with the observed data? You may assume that quicksort always selects the last element as the pivot.
    A) Sort A is insertion sort, sort B is mergesort, sort C is quicksort
    B) Sort A is insertion sort, sort B is heapsort, sort C is bubble sort
    C) Sort A is insertion sort, sort B is quicksort, sort C is mergesort
    D) Sort A is selection sort, sort B is heapsort, sort C is quicksort
    E) Sort A is selection sort, sort B is quicksort, sort C is bubble sort

14. What is the time complexity and auxiliary space required to run heapsort on an array of n elements?
    A) $\Theta(n)$ time, $\Theta(1)$ auxiliary space
    B) $\Theta(n\log(n))$ time, $\Theta(1)$ auxiliary space
    C) $\Theta(n\log(n))$ time, $\Theta(n)$ auxiliary space
    D) $\Theta(n^2)$ time, $\Theta(1)$ auxiliary space
    E) $\Theta(n^2)$ time, $\Theta(n)$ auxiliary space

15. Suppose you had the following unsorted array:

    ```
    [48, 25, 98, 41, 33, 64, 87, 38, 19]
    ```

    You decided to sort this array using heapsort with bottom-up heapify, and you took six snapshots of the array while it was being sorted. However, you forgot to keep track of the order of your snapshots, so you must look at the contents of each array to determine which came first. Given these six snapshots below, in what order were they taken?

    I. `[33, 19, 25, 38, 41, 48, 64, 87, 98]`
    II. `[48, 41, 25, 38, 33, 19, 64, 87, 98]`
    III. `[19, 33, 25, 38, 41, 48, 64, 87, 98]`
    IV. `[98, 41, 87, 38, 33, 64, 48, 25, 19]`
    V. `[87, 41, 64, 38, 33, 19, 48, 25, 98]`
    VI. `[33, 38, 25, 19, 41, 48, 64, 87, 98]`

    A) IV, V, II, VI, I, III
    B) III, I, VI, II, V, IV
    C) IV, V, VI, II, I, III
    D) IV, V, II, I, VI, III
    E) IV, V, II, VI, III, I

16. Suppose you had the following unsorted array:

    ```
    [70, 44, 69, 40, 42, 37, 46, 39]
    ```

    You decided to sort this array using heapsort, and you took four snapshots of the array while it was being sorted. However, you somehow ended up with five snapshots of this array — and one of them does not belong! The five snapshots are shown below. Which snapshot was **NOT** taken while the array was being sorted using heapsort?

    I. `[39, 40, 37, 42, 44, 46, 69, 70]`
    II. `[37, 44, 39, 40, 42, 46, 69, 70]`
    III. `[39, 44, 46, 40, 42, 37, 69, 70]`
    IV. `[42, 40, 39, 37, 44, 46, 69, 70]`
    V. `[44, 42, 39, 40, 37, 46, 69, 70]`

    A) Snapshot I
    B) Snapshot II
    C) Snapshot III
    D) Snapshot IV
    E) Snapshot V

17. Consider the following implementation of `partition()`, where the last element in the array is always chosen as the pivot.

    ```
    1  int32_t partition(int32_t a[], int32_t left, int32_t right) {
    2    int32_t pivot = --right;
    3    while (true) {
    4      while (a[left] < a[pivot])
    5        ++left;
    6      while (left < right && a[right - 1] >= a[pivot])
    7        --right;
    8      if (left >= right)
    9        break;
    10     std::swap(a[left], a[right - 1]);
    11   } // while
    12   std::swap(a[left], a[pivot]);
    13   return left;
    14 } // partition()
    ```

    Suppose you had the following unsorted array:

    ```
    int32_t arr[] = {5, 7, 4, 1, 9, 8, 2, 6, 3};
    ```

    What are the contents of this array after one call to `partition(arr, 0, 9)`?

    A) `[1, 2, 3, 4, 5, 6, 7, 8, 9]`
    B) `[2, 1, 3, 5, 9, 8, 7, 6, 4]`
    C) `[2, 1, 3, 7, 9, 8, 5, 6, 4]`
    D) `[3, 1, 2, 5, 7, 4, 9, 8, 6]`
    E) None of the above

18. You are given a string of size n that only contains lowercase English letters from 'a' to 'z'. What is the time complexity of returning a string with all the characters of the original string in sorted order, if you use the most efficient algorithm? For example, given the string `"eecsisfun"`, you would want to return the string `"ceefinssu"`.
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n\log(n))$
    E) $\Theta(n^2)$

19. Which of the following comparison sorting algorithms never compare the same pair of elements more than once during the sorting process?
    I. Insertion sort
    II. Selection sort
    III. Heapsort

    A) I only
    B) II only
    C) III only
    D) I and II only
    E) I and III only

20. Consider a quicksort algorithm that always chooses the median value as the pivot of each subarray in $\Theta(1)$ time. Given an array of n distinct values, what is the worst-case time complexity of performing this quicksort algorithm to sort this array?
    A) $\Theta(1)$
    B) $\Theta(\log(n))$
    C) $\Theta(n)$
    D) $\Theta(n\log(n))$
    E) $\Theta(n^2)$

21. Which of the following statements is/are **TRUE**?
    I. Heapsort guarantees $\Theta(n\log(n))$ worst-case performance on a container that supports random access.
    II. If you pick the best pivot with every partition, the average-case time complexity of quicksort may be better than the average-case time complexity of mergesort (for sorting the same array of values).
    III. In the average case, mergesort requires more memory usage to sort an array of size n compared to quicksort.

    A) I only
    B) II only
    C) III only
    D) I and III only
    E) I, II, and III

22. Which of the following correctly depicts the minimum amount of auxiliary space required to sort an array of size n using recursive quicksort and mergesort, assuming the most efficient implementation? Include the space required for stack frames.
    A) Quicksort: $\Theta(1)$, Mergesort: $\Theta(1)$
    B) Quicksort: $\Theta(1)$, Mergesort: $\Theta(n)$
    C) Quicksort: $\Theta(\log(n))$, Mergesort: $\Theta(1)$
    D) Quicksort: $\Theta(\log(n))$, Mergesort: $\Theta(n)$
    E) Quicksort: $\Theta(n)$, Mergesort: $\Theta(n)$

23. Which of the following sorting algorithms is best suited for sorting a small array of integers with relatively few inversions?
    A) Insertion sort
    B) Selection sort
    C) Quicksort
    D) Mergesort
    E) Both (C) and (D)

24. How many inversions exist in this array: [0, 2, 8, 1, 3, 7, 6] (assume ascending order is the desired sorting order)?
    A) 2
    B) 3
    C) 4
    D) 5
    E) 6

25. What is the maximum number of inversions that can exist in an array of size 7?
    A) 14
    B) 15
    C) 20
    D) 21
    E) 28

26. Suppose you are given an unsorted array with mystery values, as shown. Three of these mystery values are denoted as a, b, and c.

```
[..., a, ..., b, ..., c, ..., 17]
```

You prepare to quicksort this array in ascending order by selecting 17 as your first pivot. After completing the first partition, you notice that b ended up at index 0 of the array, a ended up at index 1 of the array, 17 ended up at index 2 of the array, and c ended up at the last position in the array, as shown.

```
[b, a, 17, ..., c]
```

Knowing this, which of the following statements is/are **TRUE**?

I. The value of b must be less than or equal to a.
II. The value of $a+b$ must be less than 35.
III. c must be the largest value in the array.

A) II only
B) III only
C) I and II only
D) II and III only
E) I, II, and III

27. You have a spreadsheet with post data from the Spring 2018 EECS 281 Piazza site. A subset of the data is shown below:

Post ID	View Count	Subject Line
23	225	Lecture schedule and timing
57	92	Study Group for Lab, Project, and Exams
79	115	Stack vs. Queue
140	194	Style and Organization Recommendation
232	63	Use of Classes for Project 1
519	78	WHERE IS THE LAB SLIDES?

You insert these posts into a vector in this order and index sort them on the number of views, as shown by the following comparator:

```
1  struct Post { int32_t id; int32_t views; std::string subject; };
2
3  class IdxSort {
4    const std::vector<Post>& _posts;
5  public:
6    IdxSort(const std::vector<Post>& temp) : _posts(temp) {}
7    bool operator()(size_t i, size_t j) const {
8      return _posts[i].views < _posts[j].views;
9    } // operator()()
10 };
11
12 void index_sort(const std::vector<Post>& posts) {
13   IdxSort idx_comp(posts);
14   std::vector<int32_t> indices(posts.size());
15   std::iota(indices.begin(), indices.end(), 0);
16   std::sort(indices.begin(), indices.end(), idx_comp);
17 } // index_sort
```

What are the contents of `indices` after the index sort on line 16 is complete?

A) `0 2 1 3 4 5`
B) `5 4 3 1 2 0`
C) `4 5 1 2 3 0`
D) `0 3 2 1 5 4`
E) None of the above

28. Which of the following statements about selection sort is/are **TRUE**?

I. Selection sort only needs to perform $\Theta(n)$ comparisons if the provided array is nearly sorted.
II. Selection sort runs in $\Theta(n\log(n))$ time for containers that support random access, and $\Theta(n^2)$ time for containers that do not.
III. Selection sort is useful if you want to minimize the total number of swaps performed during the sorting process.

A) I only
B) III only
C) I and II only
D) I and III only
E) I, II, and III

29. For which of the following reasons would it be preferable to use heapsort instead of quicksort to sort an array of n distinct values?

    I. You want to ensure that the relative ordering of equal elements is maintained before and after the sort.
    II. You want to ensure that the sort will take no worse than $\Theta(n \log(n))$ time.
    III. You want to ensure that the sort can be done with $\Theta(1)$ auxiliary space.

    A) I only
    B) II only
    C) III only
    D) II and III only
    E) I, II, and III

For questions 30-34, suppose you had the following unsorted array:

```
[22, 9, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47]
```

30. A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

```
[9, 13, 22, 52, 66, 74, 28, 59, 71, 11, 35, 47]
```

    which of the following sorts could have been run on this array?
    A) Bubble sort
    B) Insertion sort
    C) Selection sort
    D) Quicksort
    E) Mergesort

31. A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

```
[9, 11, 13, 22, 28, 74, 66, 59, 71, 35, 52, 47]
```

    which of the following sorts could have been run on this array?
    A) Bubble sort
    B) Insertion sort
    C) Selection sort
    D) Quicksort
    E) Mergesort

32. A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

```
[22, 9, 13, 11, 35, 28, 47, 59, 71, 66, 52, 74]
```

    which of the following sorts could have been run on this array?
    A) Bubble sort
    B) Insertion sort
    C) Selection sort
    D) Quicksort
    E) Mergesort

33. A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

```
[9, 13, 22, 28, 52, 59, 35, 66, 11, 47, 71, 74]
```

    which of the following sorts could have been run on this array?
    A) Bubble sort
    B) Insertion sort
    C) Selection sort
    D) Quicksort
    E) Mergesort

34. A snapshot is taken during execution of a sorting algorithm. If the snapshot of the array is:

```
[9, 13, 22, 28, 52, 59, 66, 74, 71, 35, 11, 47]
```

    which of the following sorts could have been run on this array?
    A) Bubble sort
    B) Insertion sort
    C) Selection sort
    D) Quicksort
    E) Mergesort

35. You are given two snapshots of an array during the execution of a sorting algorithm, where the first snapshot was taken before the second:

    Snapshot 1: `[13, 11, 12, 14, 17, 21, 15, 18, 22, 16, 19, 20]`
    Snapshot 2: `[11, 12, 13, 14, 17, 21, 15, 18, 22, 16, 19, 20]`

    Which of the following sorts could have possibly been run on this array? *Select all that apply.* Note: for quicksort, do not restrict yourself to the implementation where the last element is always chosen as the pivot; you may consider an implementation where any value could have been chosen as the pivot, and the selected pivot could have been swapped to either the front or back of the array before partitioning.

    A) Bubble sort
    B) Insertion sort
    C) Selection sort
    D) Quicksort
    E) Mergesort

Chapter 14 Exercise Solutions

1. **The correct answer is (B).** The worst-case time complexity of quicksort on an array of size n is $\Theta(n^2)$, and not $\Theta(n\log(n))$. This happens if you get unlucky with your pivot choice and end up with the smallest or largest elements as your pivot at every step, which causes the quicksort recurrence to become $T(n) = T(n-1) + n$, which evaluates to $\Theta(n^2)$.

2. **The correct answer is (A).** Of the 5 sorting algorithms, insertion sort is adaptive, which allows it to perform better on nearly sorted input.

3. **The correct answer is (B).** If the registrar has to sort a list of waitlisted students by credit hours to determine waitlist priority, such a sort cannot disturb the arrival order due to the possibility of ties. Thus, the registrar needs a sort that is stable, as stable sorts can preserve the relative order of students when there are duplicates present. Selection sort is the only sort provided that is not stable.

4. **The correct answer is (D).** Option A is false because bubble sort can be implemented to be both adaptive and stable (see section 14.2 for this exact implementation). Option B is false because insertion sort can be done in-place without any additional auxiliary space. Option C is false because selection will still need to perform $\Theta(n^2)$ comparisons, regardless of what the input is (for this reason, option D is true, since these comparisons represent a majority of the work done while running a selection sort).

5. **The correct answer is (D).** Both students require a sort that has a worst-case time complexity of $\Theta(n\log(n))$, so quicksort is not an option. Of the remaining comparison sorts, both heapsort and mergesort can be done in $\Theta(n\log(n))$ worst-case time. Mergesort is stable, so it would allow Daniel to keep the relative order of elements in his array intact. Heapsort is not stable, so it would not work for Daniel. On the other hand, heapsort can be done in-place, which would work for Ryan (mergesort would not, since it requires $\Theta(n)$ auxiliary space on an array). Thus, Daniel should use mergesort, and Ryan should use heapsort.

6. **The correct answer is (D).** Non-adaptive sorting algorithms may be simpler to implement than adaptive algorithms, as they do not have to exhibit different behavior depending on the outcomes of comparisons. However, this inability to change their operations depending on the input may cause them to run slower in special situations, such as sorting a nearly-sorted array.

7. **The correct answer is (E).** The best external sorts are divide-and-conquer sorting algorithms, which allow the input to be divided across multiple workers, individually sorted, and then merged together. Of the provided sorts, mergesort falls into this category.

8. **The correct answer is (C).** Mergesort is the only one of the provided sorts that is stable. In addition, selection sort and quicksort have a worst-case time complexity of $\Theta(n^2)$, and heapsort's $\Theta(n\log(n))$ time complexity relies on random access, which linked lists do not support.

9. **The correct answer is (C).** Statements I and II are true. Mergesort does require an auxiliary array of the same size when sorting the original array, and the fastest possible comparison sort has a worst-case time complexity no better than $\Theta(n\log(n))$ (see section 14.8.2 for an explanation why). Statement III is false because quicksort's better time complexity does not mean that it will always sort faster, but rather that its performance degrades slower as input size grows (also, the worst-case time complexity of quicksort is still $\Theta(n^2)$, which matches bubble sort).

10. **The correct answer is (A).** The best choice for the pivot is the median, which lets you partition the input into two evenly sized subarrays. In this case, the median value is 43.

11. **The correct answer is (B).** The recurrence relation for mergesort is $T(n) = 2T(n/2) + \Theta(n)$, since it recursively calls itself twice with half the input size, then calls `merge()`, which takes linear time. However, since your friend incorrectly implemented the algorithm so that the merging step takes $\Theta(n^2)$ time, the recurrence relation now becomes $T(n) = 2T(n/2) + \Theta(n^2)$. Using the Master Theorem where $a = 2$, $b = 2$, and $c = 2$, we see that $a < b^c$, which implies that the time complexity of this recurrence is $\Theta(n^c) = \Theta(n^2)$.

12. **The correct answer is (E).** Quicksort A is the standard version of quicksort that was discussed in this chapter, with an average-case time complexity of $\Theta(n\log(n))$ and a worst-case time complexity of $\Theta(n^2)$. Quicksort B always chooses the worst-case pivot, so the performance of quicksort B will always be the worst-case of $\Theta(n^2)$. Thus, the average-case time complexity of quicksort A is better than that of quicksort B, but their worst-case time complexities are the same.

13. **The correct answer is (E).** We can see from the data that sort A performs roughly the same regardless of whether the array is unsorted, nearly sorted, or fully sorted. This indicates that sort A is not adaptive; this eliminates options (A), (B), and (C), since insertion sort is adaptive. Sort B performs the best on unsorted arrays, while sort C performs the best on sorted arrays. This indicates that sort C is adaptive, since it is able to use the fact that its array is sorted to improve its performance — of the two remaining options, bubble sort makes the most sense for sort C. This leaves sort B as quicksort, which makes sense given that the quicksort algorithm used always selects the last element as the pivot; for nearly sorted arrays, quicksort will tend to select bad pivots with each partition (since it will pick pivot values that are close to the largest within each subarray rather than the median), which causes the performance to degenerate from $\Theta(n\log(n))$ to $\Theta(n^2)$.

14. **The correct answer is (B).** The process of fixing down takes $\Theta(\log(n))$ time, and it is done $n-1$ times during the heapsort process. Thus, the time complexity of heapsort is $\Theta(n\log(n))$. Heapsort can be done in-place, so it requires no additional memory.

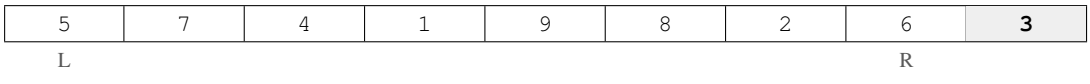
15. **The correct answer is (E).** The process of heapsort is shown below (bold = element fixed in the heap):

Step	Heap	Snapshot
Start	[48, 25, 98, 41, 33, 64, 87, 38, 19]	
Turn into a max heap	[98, 41, 87, 38, 33, 64, 48, 25, 19]	(iv)
Swap top and bottom	[19, 41, 87, 38, 33, 64, 48, 25, **98**]	
Call fix down to fix the heap	[87, 41, 64, 38, 33, 19, 48, 25, **98**]	(v)
Swap top and bottom	[25, 41, 64, 38, 33, 19, 48, **87**, **98**]	
Call fix down to fix the heap	[64, 41, 48, 38, 33, 19, 25, **87**, **98**]	
Swap top and bottom	[25, 41, 48, 38, 33, 19, **64**, **87**, **98**]	
Call fix down to fix the heap	[48, 41, 25, 38, 33, 19, **64**, **87**, **98**]	(ii)
Swap top and bottom	[19, 41, 25, 38, 33, **48**, **64**, **87**, **98**]	
Call fix down to fix the heap	[41, 38, 25, 19, 33, **48**, **64**, **87**, **98**]	
Swap top and bottom	[33, 38, 25, 19, **41**, **48**, **64**, **87**, **98**]	(vi)
Call fix down to fix the heap	[38, 33, 25, 19, **41**, **48**, **64**, **87**, **98**]	
Swap top and bottom	[19, 33, 25, **38**, **41**, **48**, **64**, **87**, **98**]	(iii)
Call fix down to fix the heap	[33, 19, 25, **38**, **41**, **48**, **64**, **87**, **98**]	(i)
Swap top and bottom	[25, 19, **33**, **38**, **41**, **48**, **64**, **87**, **98**]	
Call fix down to fix the heap	[25, 19, **33**, **38**, **41**, **48**, **64**, **87**, **98**]	
Swap top and bottom	[**19**, **25**, **33**, **38**, **41**, **48**, **64**, **87**, **98**]	

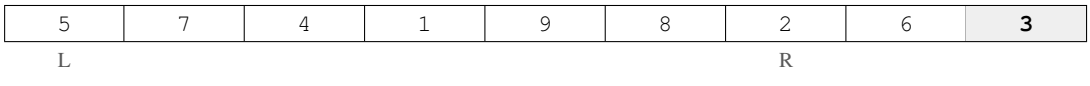
16. **The correct answer is (A).** The process of heapsort is shown below (bold = element fixed in the heap):

Step	Heap	Snapshot
Start (already a max heap)	[70, 44, 69, 40, 42, 37, 46, 39]	
Swap top and bottom	[39, 44, 69, 40, 42, 37, 46, **70**]	
Call fix down to fix the heap	[69, 44, 46, 40, 42, 37, 39, **70**]	
Swap top and bottom	[39, 44, 46, 40, 42, 37, **69**, **70**]	(iii)
Call fix down to fix the heap	[46, 44, 39, 40, 42, 37, **69**, **70**]	
Swap top and bottom	[37, 44, 39, 40, 42, **46**, **69**, **70**]	(ii)
Call fix down to fix the heap	[44, 42, 39, 40, 37, **46**, **69**, **70**]	(v)
Swap top and bottom	[37, 42, 39, 40, **44**, **46**, **69**, **70**]	
Call fix down to fix the heap	[42, 40, 39, 37, **44**, **46**, **69**, **70**]	(iv)
Swap top and bottom	[37, 40, 39, **42**, **44**, **46**, **69**, **70**]	
Call fix down to fix the heap	[40, 37, 39, **42**, **44**, **46**, **69**, **70**]	
Swap top and bottom	[39, 37, **40**, **42**, **44**, **46**, **69**, **70**]	
Call fix down to fix the heap	[39, 37, **40**, **42**, **44**, **46**, **69**, **70**]	
Swap top and bottom	[**37**, **39**, **40**, **42**, **44**, **46**, **69**, **70**]	

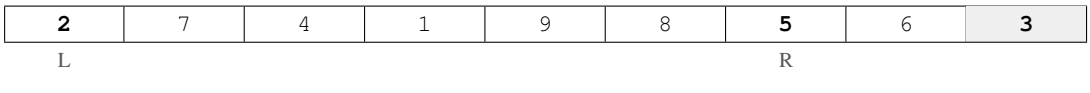
17. **The correct answer is (C).** We select 3 as the pivot and set a left and right pointer to begin partitioning:



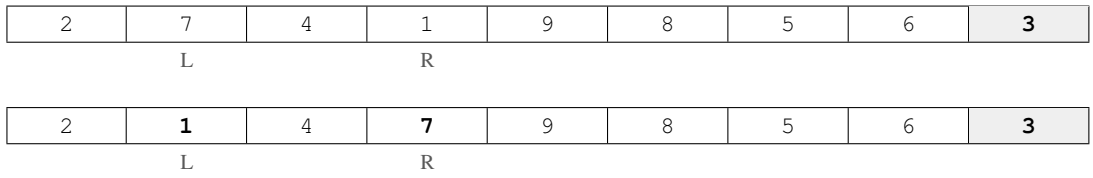
We keep on incrementing the left pointer until we find a value greater than the pivot value of 3 (in this case, we already have such an element in 5). Then, we decrement the right pointer until we find a value less than 3 (the first of which is 2).



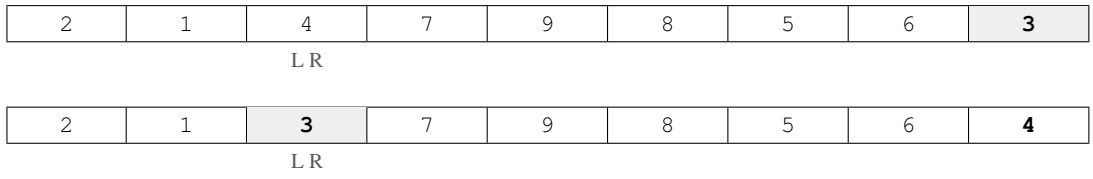
Left and right are not pointing to the same value, so we swap the two elements:



We then increment the left pointer until we find a value greater than 3; this lands the left pointer at 7. We then decrement the right pointer until we find a value less than 3; this lands the right pointer at 1. Since left and right are not the same value, we swap the elements:



We then increment the left pointer until we find a value greater than 3; this lands the left pointer at 4. We then decrement the right pointer until we find a value less than 3 — however, this cases the left and right pointers to point to the same value. When this happens, we swap the pivot with the element at the position where the two pointers meet. Thus, 3 and 4 swap places to produce our final array:



18. **The correct answer is (C).** The number of possible characters in the string is limited to 26 possible characters: 'a' to 'z'. Because of this, we can use counting sort to sort the string in linear time.

19. **The correct answer is (A).** Only insertion sort never compares the same pair of elements more than once during the sorting process, since each element is only ever compared with the values that come before it. Selection sort may compare the same pair of elements more than once: for example, given [3, 2, 1], selection sort would compare 2 and 3 on the first pass when determining the smallest element to place at index 0, then it would swap 3 and 1, and then would compare 2 and 3 again on the second pass to determine the next smallest element to place at index 1. Heapsort may also compare the same pair of elements more than once since you cannot guarantee the organization of elements within the heap throughout the sorting process: for instance, in the example shown at the beginning of section 14.5, the values of 46 and 17 are compared more than once.

20. **The correct answer is (C).** If the quicksort algorithm can always choose the median value as the pivot in $\Theta(1)$ time, then each partition will always split its subarray in half. This gives us the following recurrence relation for quicksort in all cases:

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ 2T(n/2) + n, & \text{if } n > 1 \end{cases}$$

This evaluates to $\Theta(n \log(n))$; essentially, if the median is chosen as the pivot every time, then the worst-case of our original quicksort implementation will never happen.

21. **The correct answer is (D).** Statement I is true because heapsort guarantees $\Theta(n \log(n))$ time if given random access, since it involves a $n-1$ calls to fix down, which takes $\Theta(\log(n))$ time. Statement II is false because the average-case time complexity of mergesort is $\Theta(n \log(n))$, and it is not possible for quicksort to be better than that as a comparison sort. Statement III is true, since the auxiliary space required by quicksort on an array is $\Theta(\log(n))$, compared to $\Theta(n)$ for mergesort.

22. **The correct answer is (D).** The auxiliary space used by quicksort is $\Theta(\log(n))$ for the stack frames required (if we always make sure we recurse into the larger subarray last to ensure it is tail recursive). The auxiliary space used by mergesort is $\Theta(n)$ for the auxiliary array needed to store the sorted/merged output.

23. **The correct answer is (A).** An array with few inversions is one that is nearly sorted, as there are few pairs of elements that are out of order relative to each other. Insertion sort is the only adaptive sorting algorithm among the ones provided, so it can perform better for a small, nearly sorted array.

24. **The correct answer is (E).** There are six inversions in this array: (2, 1), (8, 1), (8, 3), (8, 7), (8, 6), and (7, 6).

25. **The correct answer is (D).** The maximum number of inversions occurs if the array is in reverse sorted order, for which every pair of values is an inversion. For an array of size 7, there are 6 inversions that start with the first value, 5 inversions that start with the second value, 4 inversions that start with the third value, 3 inversions that start with the fourth value, 2 inversions that start with the fifth value, and 1 inversion that starts with the sixth value. This comes out to $6+5+4+3+2+1 = 21$ inversions. In general, the maximum number of possible inversions in an array of size n is $(n-1) + (n-2) + \ldots + 1$. To illustrate with another example, the inversions in the array [4, 3, 2, 1] are (4, 3), (4, 2), (4, 1), (3, 2), (3, 1), and (2, 1) — there are total of $3 + 2 + 1 = 6$ inversions for an array of size 4.

26. **The correct answer is (A).** The only thing we can conclude after the first partition is that the first pivot ends up in the correct position in sorted order, and that all values to the left are $\leq$ pivot, and all values to the right are $\geq$ pivot. You cannot conclude anything about the order of values in these two subarrays outside of the pivot value. In this case, we know that 17 is the third smallest value in the array since it ended up at index 2 after the first partition, and that b and a are each no greater than 17, and c is no less than 17. Thus, the only statement we can guarantee is statement II, since $a+b$ cannot be greater than 34 if neither a nor b can be larger than 17.

27. **The correct answer is (C).** The index sort sorts the Piazza posts in order of view count, where the index of the lowest viewed post comes first and the index of the highest viewed post comes last. In this case, the sorted order of views is post 232 (index 4) → post 519 (index 5) → post 57 (index 1) → post 79 (index 2) → post 140 (index 3) → post 23 (index 0).

28. **The correct answer is (B).** Only statement III is true, since selection sort only requires $n-1$ swaps in the worst-case, when given an input size of n (one swap per iteration). Statement I is false since selection sort will still need to perform $\Theta(n^2)$ comparisons ($\Theta(n)$ comparisons per iteration to determine which value is the next smallest), and statement II is false because selection sort takes $\Theta(n^2)$ time regardless of whether random access is supported.

29. **The correct answer is (D).** Statements II and III are true: heapsort can be done in $\Theta(n\log(n))$ worst-case time and $\Theta(1)$ auxiliary space, while quicksort requires $\Theta(n^2)$ worst-case time and $\Theta(\log(n))$ auxiliary space. Statement I is false because heapsort is not stable.

30. **The correct answer is (E).** You can see that the elements can be split into sorted subsections:

    ```
    [9, 13, 22]  [52, 66, 74]  [28, 59, 71]  [11, 35, 47]
    ```

    This indicates the array is in the process of being sorted with mergesort.

31. **The correct answer is (C).** The process of selection sort is shown below; you can see that the smallest values are swapped with the values that were at the beginning of the original array.

    ```
    [22, 9, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 22, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 11, 13, 52, 66, 74, 28, 59, 71, 35, 22, 47]
    [9, 11, 13, 22, 66, 74, 28, 59, 71, 35, 52, 47]
    [9, 11, 13, 22, 28, 74, 66, 59, 71, 35, 52, 47]
    ```

32. **The correct answer is (D).** Notice how 47 acts as a pivot, with numbers to the left of 47 being less than 47, and numbers to the right of 47 being greater than 47. This hints that the array is being sorted with quicksort (and you can prove this by walking through the first few steps):

    ```
    [22, 9, 13, 11, 35, 28, 47, 59, 71, 66, 52, 74]
    ```

33. **The correct answer is (A).** Notice that the larger elements are bubbling to the right side of the array. This hints that bubble sort is being executed on this array. The following steps show that this is true:

    ```
    [22, 9, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 22, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 28, 74, 59, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 28, 59, 74, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 28, 59, 71, 74, 35, 11, 47]
    [9, 13, 22, 52, 66, 28, 59, 71, 35, 74, 11, 47]
    [9, 13, 22, 52, 66, 28, 59, 71, 35, 11, 74, 47]
    [9, 13, 22, 52, 66, 28, 59, 71, 35, 11, 47, 74]
    [9, 13, 22, 52, 28, 66, 59, 71, 35, 11, 47, 74]
    [9, 13, 22, 52, 28, 59, 66, 71, 35, 11, 47, 74]
    [9, 13, 22, 52, 28, 59, 66, 35, 71, 11, 47, 74]
    [9, 13, 22, 52, 28, 59, 66, 35, 11, 71, 47, 74]
    [9, 13, 22, 52, 28, 59, 66, 35, 11, 47, 71, 74]
    [9, 13, 22, 28, 52, 59, 66, 35, 11, 47, 71, 74]
    [9, 13, 22, 28, 52, 59, 35, 66, 11, 47, 71, 74]
    ```

    While this variation of bubble sort bubbles larger values to the right side of the array, this is not the only possible implementation of this sorting algorithm. It is also valid for a bubble sort implementation to bubble the smaller values to the left side of the array.

34. **The correct answer is (B).** For insertion sort, you will notice that the values are moved to the front of the array in a manner that incrementally expands the portion of the array that is sorted from left to right. The process of insertion sort is shown below; the underlined values represent the portion of the array that is sorted, and the bolded value represents the new value that was added to the sorted subarray.

    ```
    [22, 9, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 22, 13, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 13, 22, 52, 66, 74, 28, 59, 71, 35, 11, 47]
    [9, 13, 22, 28, 52, 66, 74, 59, 71, 35, 11, 47]
    [9, 13, 22, 28, 52, 59, 66, 74, 71, 35, 11, 47]
    ```

35. **The correct answers are (A), (B), (C), (D), and (E).** All of the provided sorts could have been used. For bubble sort, the second snapshot could have been taken after 13 bubbled past 12 to its correct position. For insertion sort, the second snapshot could have been taken after 12 was shifted before 13 when considering the first three elements. For selection sort, the second snapshot could have been taken after 13 was first swapped with 11 on the first pass, and then 12 on the second pass. For quicksort, 12 could have been chosen as the first pivot and then swapped to the front, and the second snapshot could have been taken after the other elements are partitioned. For mergesort, the elements in the second snapshot can be split into sorted subarrays of size 3.