Note: these were written a while back (e.g. over a year ago), so if there are any clarity issues (or mistakes), please let me know! Also, feel free to ask a question on Piazza if anything is unclear; I'm sure I can probably come up with a better explanation for some of these questions right now than I did back then. Good luck studying!

1. **The correct answer is (A).** None of the statements are true. The asymptotic efficiency of algorithms deals with how runtime increases with the size of the input as the size of the input increases without bound (i.e. its scalability). For instance, as the size of the input grows, we can reason that a $\Theta(n^2)$ algorithm will see its runtime increase much more drastically than a $\Theta(n \log n)$ algorithm. However, it does not guarantee that an algorithm with a $\Theta(n^2)$ time complexity will always run faster than an algorithm with a $\Theta(n \log n)$ complexity – there are many other factors at play: for instance, the $\Theta(n^2)$ algorithm may have been run in a best-case scenario, or the constant associated with the $\Theta(n \log n)$ algorithm may be much larger. The same explanation also applies to algorithms with other complexities.

5. **The correct answer is (D).** A $\Theta(n^3 \log n)$ algorithm has a larger time complexity than a $\Theta(n^3)$ algorithm due to the presence of an additional $\log n$ that is multiplied with n^3. A $\Theta(n!)$ algorithm has a larger time complexity than a $\Theta(2^n)$ algorithm, and a $\Theta(n^n)$ algorithm has a larger complexity than both $\Theta(2^n)$ and $\Theta(n!)$ algorithms. You can determine this by testing a few values of n, as shown below.

n	2	4	8	16
2^n	4	16	256	65,536
$n!$	2	24	40,320	20,922,789,888,000
n^n	4	256	16,777,216	18,446,744,073,709,551,616

7. **The correct answer is (D).** Both the for loops on lines 3 and 4 run on the order of n times given the value of n. Because the inner loop runs on the order of n times inside the outer loop, and the outer loop itself runs n times, the operation in the inner loop (incrementing `count`) runs a total of n^2 times (running n times in the inner loop, which itself runs n times in the outer loop), for an overall complexity of $\Theta(n^2)$.

8. **The correct answer is (A).** The while loop on line 3 cuts the value of n down by half with each iteration, so the loop runs on the order of $\log_2 n$ times. Thus, the overall complexity is $\Theta(\log n)$.

9. **The correct answer is (C).** The innermost loop on line 6 runs from 0 to n in increments of $\sqrt{n}$, so the operations on lines 7 and 8 run on the order of $n/\sqrt{n}$, or $\sqrt{n}$ times. The middle loop on line 5 runs from 1 to n, but on each iteration, the value is doubled. Thus, this loop runs on the order of $\log_2 n$ times. The outermost loop on line 4 runs from $n/2$ to n in increments of 1, so the loop runs $n/2$ times, or on the order of n (remember we can disregard constants). Putting everything together, the constant work operations on lines 7 and 8 run $\sqrt{n}$ times in the innermost loop, which itself runs $\log n$ times in the middle loop, which itself runs n times in the outermost loop. We can get the overall complexity by multiplying the individual loop iterations together: $\Theta(\sqrt{n} \times \log n \times n)$, or $\Theta(n^{3/2} \log n)$.

12. **The correct answer is (D).** If the input size is too small, the relationship may not be revealed. Plotting out runtimes is most revealing when the input size is large.

13. **The correct answer is (C).** Your friend's program ran better than expected, so it could not have exposed worst-case behavior.

16. **The correct answer is (C).** If a number has a negative counterpart, the two together must sum to zero. Thus, you can simply sum the entire array (an $\Theta(n)$ procedure): values with negative counterparts will cancel out to zero, and you'll end up with the number without a negative counterpart.

17. **The correct answer is (B).** If the array did have the missing number, the sum of the array would have been

$$\frac{n(n+1)}{2}$$

where n is the size of the array. However, since one value is missing, the difference between the expected sum and the actual sum must be the value that is missing. Thus, the missing number can be identified by simply summing the array and subtracting the result from the expression above, which can be done in $\Theta(n)$ time and $\Theta(1)$ memory.

20. **The correct answer is (D).** When you call `resize` on a vector, you actually change the number of elements it contains. On the other hand, if you call `reserve`, you only reserve space for more elements, but you do not create them. Hence, you cannot use `.push_back()` on a vector that has been resized, since you would be adding elements on top of the elements you already created using `resize` (i.e. if you want 20 elements in your vector and nothing more, resizing the vector would initialize 20 elements in the vector, and using `.push_back()` would add a 21st element instead of modifying the 1st element). Meanwhile, after reserving space for a vector to hold more elements, you still must add the elements themselves. For instance, if you reserve a vector to hold 20 elements, it now has the capacity to hold 20 elements, but it doesn't actually have any elements yet (similar to how an empty 5-gallon bucket *can* hold 5 gallons of water, but it does not actually hold any water). Thus, you cannot use `[]` on a vector that only has space reserved, as the elements you want to create don't exist yet. The only ways to ensure that a vector has the number of elements you want it to have are to either resize the vector and modify each element using `[]` or reserve space for the vector and add the elements using `.push_back()`.

24. **The correct answer is (E).** This problem can be solved using the substitution method:

Step #	**Recurrence Equation**	**Subproblem Solution**
1	$T(n) = 3T(n-1) + c$	$T(n-1) = 3T(n-2) + c$
2	$T(n) = 3[3T(n-2) + c] + c = 3^2 T(n-2) + 3c + c$	$T(n-2) = 3T(n-3) + c$
3	$T(n) = 3^2[3T(n-3) + c] + 3c + c = 3^3 T(n-3) + 9c + 3c + c$	$T(n-3) = 3T(n-4) + c$
…	…	…
k	$T(n) = 3^k T(n-k) + c\sum_{m=0}^{k-1} 3^m$	$T(n-k) = 3T(n-k-1) + c$
…	…	…
$n-1$	$T(n) = 3^{n-1} T(1) + c\sum_{i=0}^{n-2} 3^i = 3^{n-1} c_0 + \frac{1}{2}c(3^{n-1} - 1)$	$T(1) = c_0$

In other words, $T(n) = 3T(n-1) + c$ can be simplified to

$$T(n) = 3^{n-1}c_0 + \frac{1}{2}c(3^{n-1} - 1) = \mathbf{3^{n-1}\left(c_0 + \frac{1}{2}c\right) - 1} = \Theta(3^n)$$

You don't need to know where the $\frac{1}{2}c(3^{n-1} - 1)$ term came from, but even without this, you should recognize that the 3^{n-1} term dominates via simple substitution. $\Theta(3^n)$ is the only viable solution.

25. **The correct answer is (D).** This problem can be solved using the Master Theorem. Here, a is 4, b is 2, and c is the coefficient of the dominating term, or 2 in the case of n^2. Since $a = b^c$, we can conclude that $T(n) = \Theta(n^c \log n) = \Theta(n^2 \log n)$.

26. **The correct answer is (B).** This problem can be solved using the Master Theorem. Here, a is 5, b is 25, and c is ½. Since $a = b^c$, we can conclude that $T(n) = \Theta(n^c \log n) = \Theta(n^{1/2} \log n)$.

27. **The correct answer is (E).** This problem can be solved using the Master Theorem. Here, a is 729, b is 9, and c is the coefficient of the dominating term, or 4/3 in this case. Since $a > b^c$, we can conclude that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_9 729}) = \Theta(n^3)$.

29. **The correct answer is (E).** Considering the `pie` function as $T(n)$, running `pie(n/7)` on line 6 would be $T(n/7)$. The function `cake(n)`, a $\log n$ operation, is run n times in the inner loop on line 11, which itself is run n^2 times on line 10 (since `cookie` is n^2). Thus, `cake(n)` runs a total of n^3 times, resulting in $n^3 \log n$. Then, `pie(n/3)` is run n times on line 17, which results in the recurrence $nT(n/3)$. Lastly, `cake` is run on the value of n^4 since `cookie` is n^2. Thus, this runs in $\log(n^4)$ time, or $4 \log n$. Sum these together.

30. **The correct answer is (C).** Since the recurrence is directly solvable by the Master Theorem, we know that the value of b (the denominator in the recurrence) must be greater than 1. Consequently, since a and c are both 1, the value of a must be less than b^c. As such, the complexity of $T(n)$ must follow the form $\Theta(n^c)$, or in this case, $\Theta(n)$.

60. **The correct answer is (A).** The time complexity of adding an element to a stack implemented with a linked list is also $\Theta(1)$; simply attach the element to the beginning of the list.

65. **The correct answer is (D).** When 203, 370, and 281 are pushed into the container, 370 is the first to be retrieved. When 280, 376, and 183 are added to the container, 376 is the first to be retrieved. After 376 is popped off, 281 is the next to be retrieved. Notice that the top value is always the largest value in the container; thus, of the options provided, the container must be a priority queue.

66. **The correct answer is (B).** When 203, 370, and 281 are pushed into the container, 281 is the first to be retrieved. When 280, 376, and 183 are added to the container, 183 is the first to be retrieved. After 183 is popped off, 376 is the next to be retrieved. Notice that the top value is always the most recent element entered into the container; thus, of the options provided, the container must be a stack.

68. **The correct answer is (D).** When you push back an element into a circular buffer, you increment the back iterator; if this back iterator ends up at the same position as the first element in the circular buffer (the front iterator), you know that your circular buffer is full!

69. **The correct answer is (A).** Darget tracks its inventory by assuming that inventory that arrives first is sold first, which is how a queue works. Paolmart tracks its inventory by assuming that inventory that arrives last is sold first, which is how a stack works.

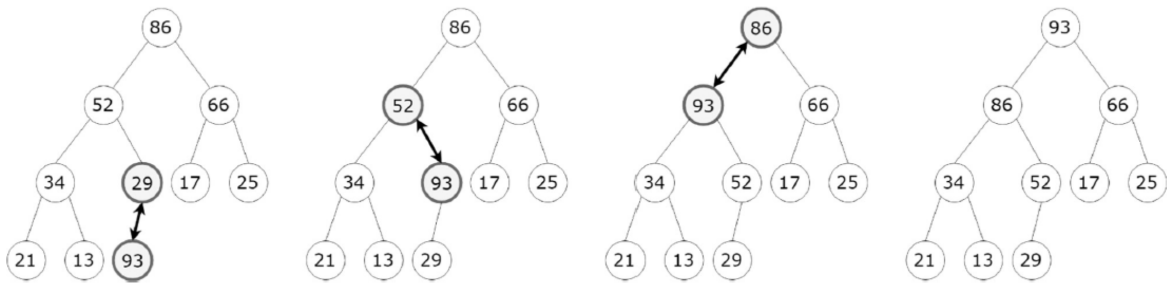
73. The answer is (D). If the comparator returns left < right, it's a less comparator, which creates a max PQ (a rule of thumb - if you were to sort a vector using the comparator, the element that ends up at the *back* after the sort is completed is the element with the greatest priority). This comparator is in the format left > right, so it is a greater comparator (or min PQ). This is a min PQ of elements based on distance from 25, so elements with a *smaller* distance to 25 (i.e. closer to 25) have the largest priority.

74. **The correct answer is (C).** Each push into a priority queue takes $\Theta(\log n)$ time, so doing this n times would result in a complexity of $\Theta(n \log n)$.

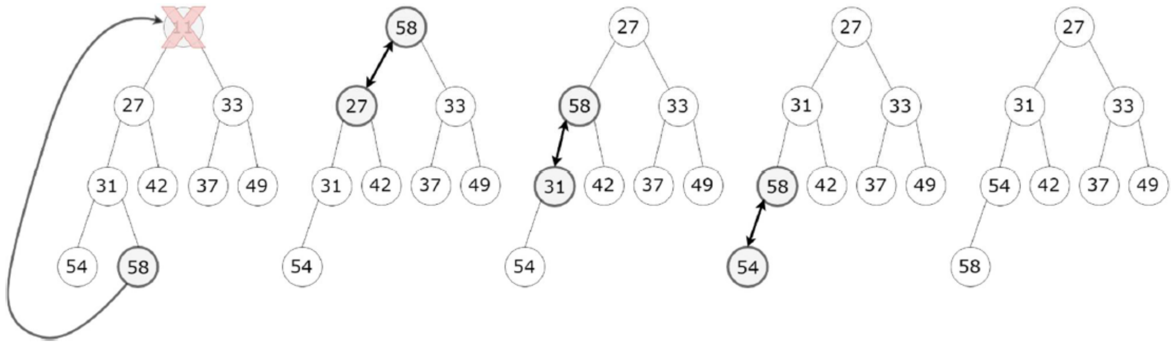
82. **The correct answer is (A).** A min-heap is valid if the children of each element (located at indices 2*i* and 2*i* + 1 using 1-indexing) are not smaller than it. Choice (B) is not a valid binary min-heap because the children of 47, 9 and 12, are smaller than 47. Choice (C) is not a valid binary min-heap because the child of 12, 11, is smaller than 12. Choice (D) is not a valid binary min-heap because a child of 59, 58, is smaller than 59. Choice (A) is valid: 13 and 8 are not smaller than 2, 16 and 13 are not smaller than 13, 10 and 40 are not smaller than 8, and 25 and 17 are not smaller than 16.

83. **The correct answer is (C).** A max-heap is valid if the children of each element (located at indices 2*i* and 2*i* + 1 using 1-indexing) are not larger than it. Choice (A) is not a valid binary max-heap because the children of 14, 25 and 27, are larger than 14. Choice (B) is not a valid binary max-heap because a child of 9, 10, is larger than 9. Choice (D) is not a valid binary max-heap because a child of 54, 56, is larger than 54. Choice (C) is valid: 24 and 24 are not larger than 33, 7 and 9 are not larger than 24, 24 and 18 are not larger than 24, 7 and 5 are not larger than 7, and 9 and 8 are not larger than 9.

86. **The correct answer is (A).** Add 93 to the end of the heap and move it up until it is in the correct position:



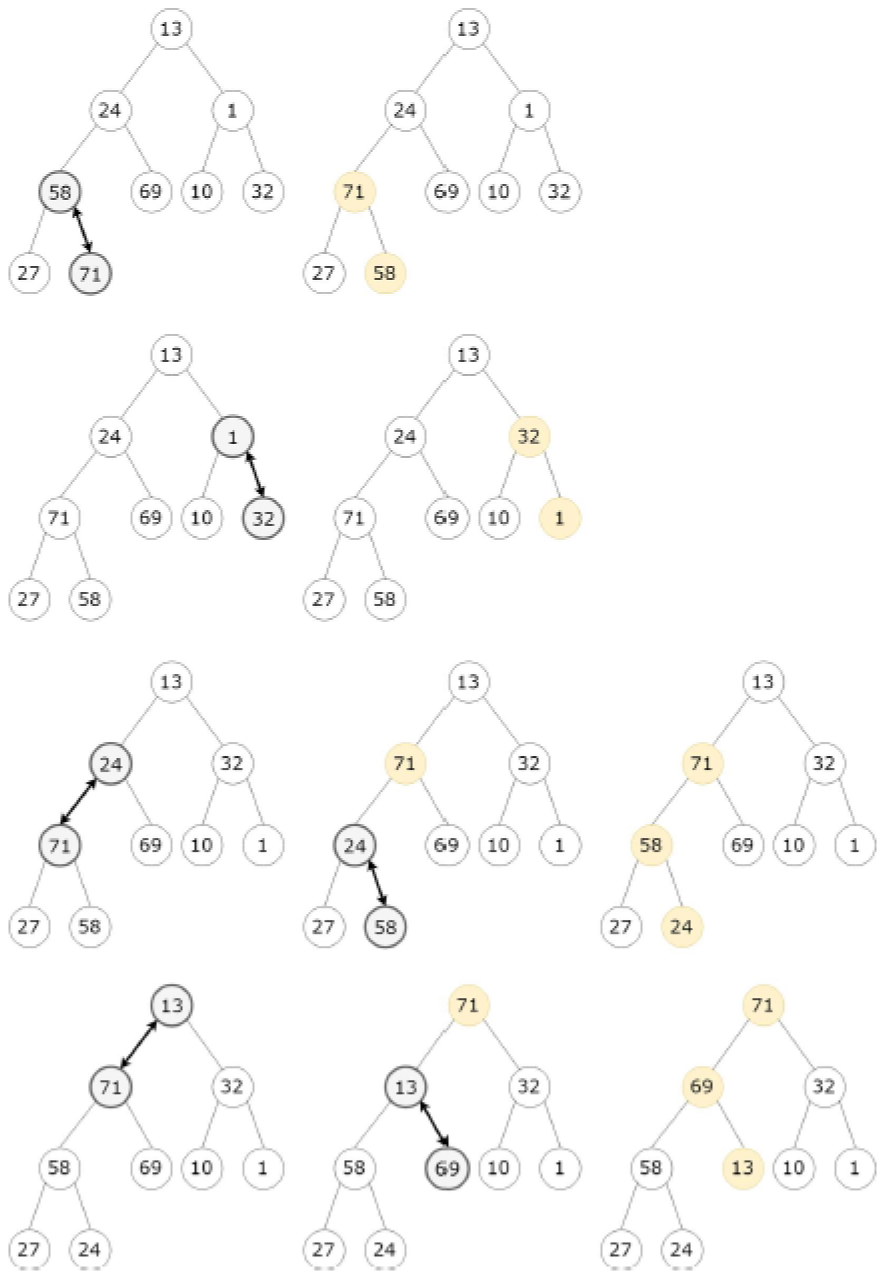
87. **The correct answer is (D).** Move 58 to the top of the heap (after deleting 11) and move it down until it is in the correct position:



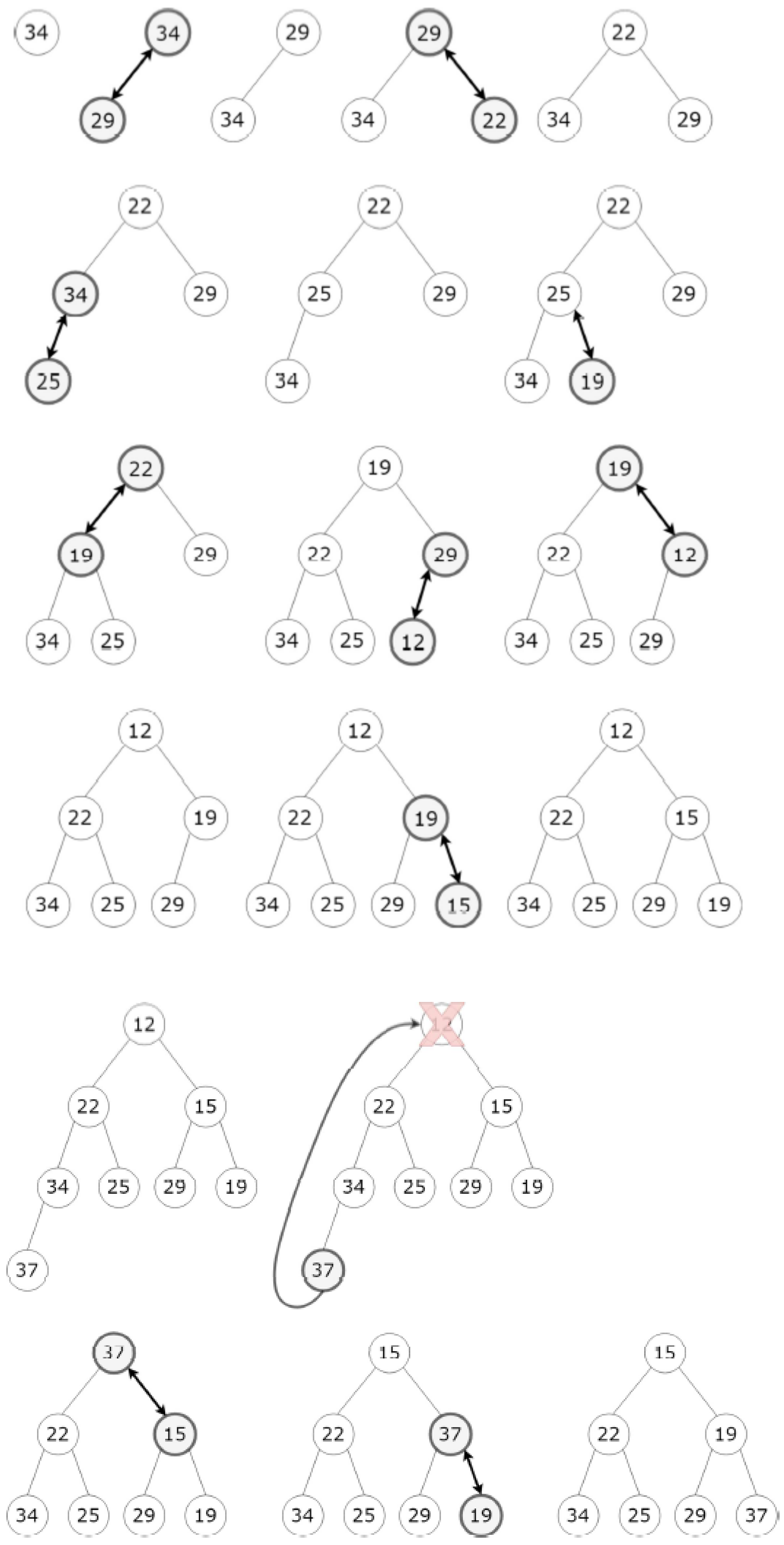
88. **The correct answer is (B).** Choice (A) is not a valid binary heap because 12 is larger than 11. Choice (C) is not a valid binary heap because it is not complete (28 only has one child but 44 has two). Choice (D) is not a valid binary heap because 44's parent and child both include 43.

89. **The correct answer is (A).** You can conduct heapify in place, using the given array of elements and `fixUp()` or `fixDown()`.

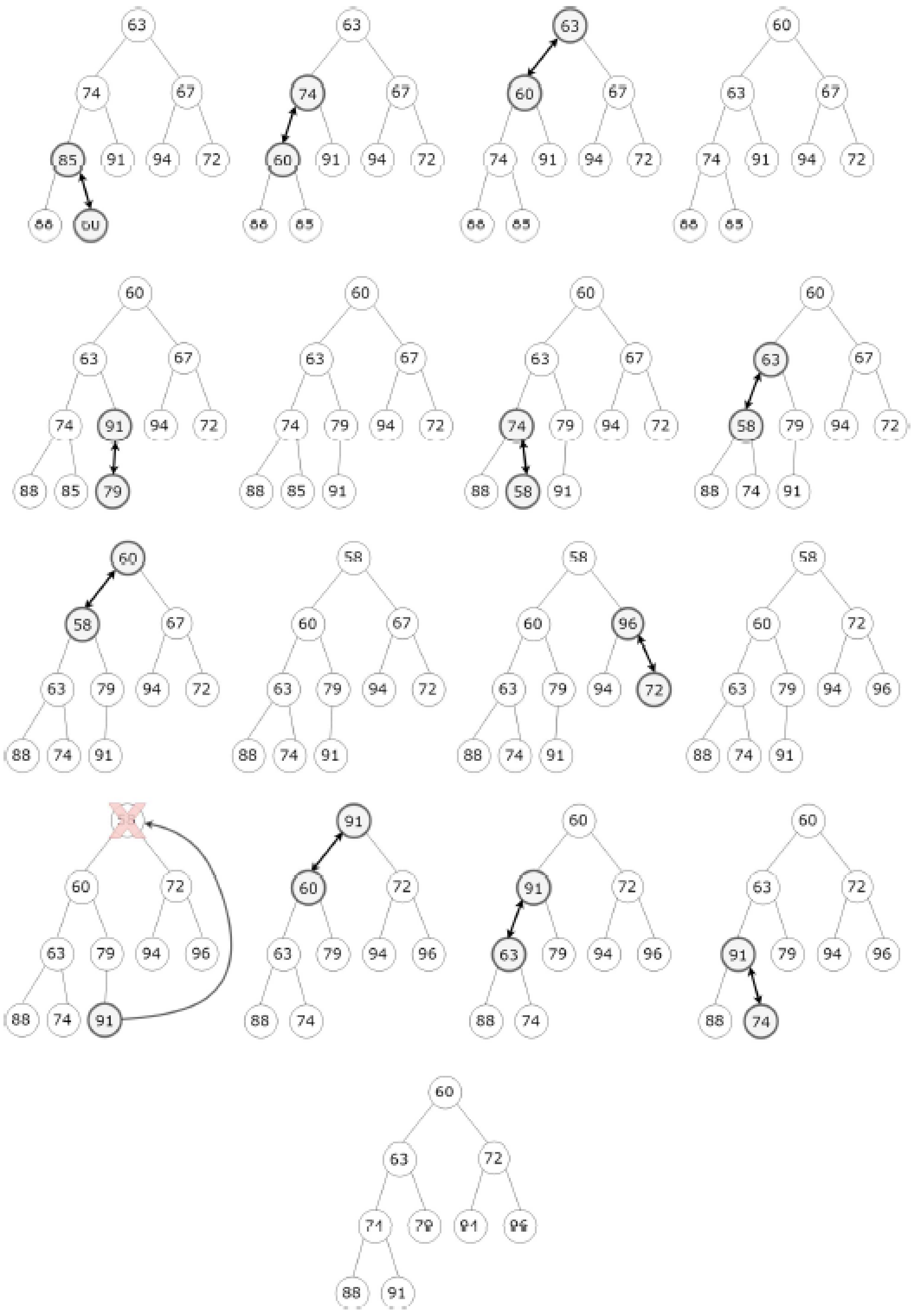
92. The correct answer is (D). Start from the bottom and move upwards, continuously calling `fixDown`:



93. The correct answer is (B). The steps are shown below:



94. The correct answer is (B). The steps are shown below:



105. The correct answer is (B). There are two disjoint sets: `{1, 3, 4, 6, 8, 9, 10}` and `{2, 5, 7}`.

106. The correct answer is (D). Using the path-compression approach, if we were to call `find()` on an element *j*, we would traverse all elements on the way to its ultimate representative *k* and change all these elements to have a representative of *k*. Since we called `find()` on element 1, we follow the chain all the way to 1's ultimate representative, or 9:

$$1 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 8 \rightarrow 10 \rightarrow 9$$

Here, all of these elements would have their representatives changed to 9.

107. The correct answer is (D). Calling `find()` on an element would not change its representative if there are no intermediaries between such an element and its ultimate representative. This is only true for element 2, as 2's representative is 7, which is its own representative. Calling `find()` on element 8 would change its representative to 9, calling `find()` on element 5 would change its representative to 7, and calling `find()` on element 4 would change its representative to 9 (as well as others down the chain to 9).

109. The correct answer is (A). Since `arr2[]` is almost sorted, the fact that `arr2[]` was sorted faster than `arr1[]` means that the sorting algorithm used must be adaptive. Of the sorts provided, only insertion sort is adaptive.

111. 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.

112. The correct answer is (E). Only choice (E) is true, as selection sort will always do on the order of n^2 comparisons regardless of the contents of the array. Choice (A) is false because bubble sort can be implemented to be both adaptive and stable. Choice (B) is false because insertion sort requires $\Theta(1)$ auxiliary space. Choice (C) is false because insertion sort with binary search would still be stable if elements are inserted after their duplicates in the array. Choice (D) is false because selection sort performs $\Theta(n^2)$ comparisons, even if it does no swaps.

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

117. Bucket sort won't be on the exam, so ignore this.

119. The correct answer is (C). Notice that Nathan Moos was inserted into the vector before Nathan Fenner. However, after the sort, Nathan Fenner appears before Nathan Moos. Because of this, we can conclude that the `mysterySort` algorithm is not stable, making selection sort the only possible option.

120. The correct answer is (A). The best choice for the pivot is the median, which is 43 in this case.

124. The correct answer is (C). The process of selection sort is shown below:

{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**, 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}

125. 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 of quicksort).

{22, 9, 13, 11, 35, 28, **47**, 59, 71, 66, 52, 74}

126. The correct answer is (A). Notice how 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}

127. **The correct answer is (B).** The process of insertion sort is shown below:

{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}

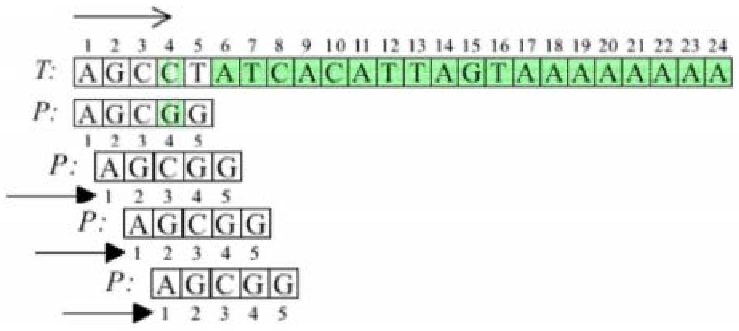
{9, 13, 22, 52, 66, **74**, 28, 59, 71, 35, 11, 47}

{9, 13, 22, **28**, 52, 66, 74, 59, 71, 35, 11, 47}

{9, 13, 22, 28, 52, **59**, 66, 74, 71, 35, 11, 47}

129. **The correct answer is (C).** The worst-case time complexity of sorting a vector of n elements is $\Theta(n \log n)$. The worst-case time complexity of comparing two strings of length m is $\Theta(m)$. The overall worst-case time complexity of sorting n strings of length m involves doing a $\Theta(m)$ comparison for every string in the vector, which results in $\Theta(mn \log n)$.

131. **The correct answer is (E).** With brute force string searching, we take the needle of length n and use it as a "window" to compare with substrings of length n within the haystack. For instance, if we had a needle of length 4 and a haystack of length 20, we would first check if the needle matches the haystack from characters 1-4, then if the needle matches with characters 2-5, then 3-6, then 4-7, etc. until we find a match. In the average case, we would be able to tell quickly if a match is found, as we expect the first letter of the needle and the haystack substring to be different if there is no match. Thus, we would check the first letter of the needle with each letter of the haystack, moving forward if a match is not found. We would only have to do approximately h comparisons when doing this, where h is the length of the haystack, so this gives us an average-case time complexity of $\Theta(h)$. However, in the worst case, we would have to check all n characters in the needle for *every* substring of length n in the haystack. Thus, for the approximately h comparisons we have to do when traversing through the haystack, we would have to do an additional n comparisons in the worst case (e.g. if we had a needle of length 4 and a haystack of length 20, we would have to compare the 4 characters of the needle with *every* haystack character from 1-4 to see if there is a match, then with *every* haystack character from 2-5, then 3-6, then 4-7, etc.). This gives us a worst-case time complexity of $\Theta(nh)$.


T: AGCCTATCACATTAGTAAAAAAAA
P: AGCGG
1 2 3 4 5 ... 24