Overview

This diagnostic test is a set of multiple-choice questions (rather than complete USACO-style coding problems) so that we can cover more topics in less time. Some questions may involve ideas or terms that you have not seen before – it's OK not to get everything right, and some problems might be good learning experiences for you! Just do the best you can with the knowledge you currently have. There is no time limit.

A Note On Big-O Notation

In case you are unfamiliar with big-O notation, you can think of it informally as describing how the running time of a solution scales with the size of the dataset that it runs on. If a piece of code runs on a list with N elements and has a running time of $O(N^2)$, for example, then doubling the length of the list will make the code take roughly four times as long to run. Notice that this notation obscures constant factors and lower-order terms; $3N^2$ and $N^2 + 7N$ are both $O(N^2)$, but the second one might be noticeably faster in practice.

Big-O notation is not used directly in the USACO, but it is important to be able to understand roughly how fast we can expect a solution to run. As a good rule of thumb, expect that your code will be able to perform around 10^8 operations in 2 seconds on the USACO servers. For example, when N can be as large as 10^5, an $O(N^2)$ solution would be too slow for the USACO time limit, so there is not much point in implementing one except maybe to pick off a few easier test cases. However, an $O(N \log N)$ solution would probably be fine.

Scoring Guide

Each question is worth 1 point.

- **16–20 questions correct:** You have an excellent foundation for CodeWOOT. We hope to see you in class!
- **11–15 questions correct:** You can succeed in CodeWOOT if you are motivated and willing to put in some extra work – particularly if you are already a mid or high Silver solver, and/or some of the misses are due to small mistakes or a lack of familiarity with some terms. Be sure to read the explanations for the questions you got wrong.
- **0–10 questions correct:** The course may be too challenging for you at the moment. We recommend practicing more with USACO problems and learning more about Bronze and Silver level algorithms and data structures.

Good luck!

Questions begin on the next page.

1. **For Questions 1-2, consider the following problem.**

> Farmer John has 26 bales of hay, each of which is labeled with a different uppercase letter of the English alphabet. He has arranged the bales in a ring so that they read `ABC` ... `Z` in clockwise order.
>
> Farmer John will give Bessie a word W (no more than N letters long, $1 \leq N \leq 10^5$) and ask her to spell it out by walking clockwise around the circle. When Bessie reaches a bale of hay, she can choose to say that bale's letter as many times as she wants (possibly zero) before moo-ving on. Bessie is done when she has said every letter in W, in the order in which those letters appear in W.
>
> Bessie starts just before bale `A` (right between `Z` and `A`, heading toward `A`). Her goal is to minimize the total number of complete cycles she makes around the ring of bales, i.e., the number of times she reaches this original starting point again. What is this minimum number?
>
> (For example, the answer for the test case `BESSIE` is 2, and the answer for the test case `FORTY` is 0.)

Suppose we write code that moves Bessie one bale at a time, and has her make as much remaining progress through the word as possible at each bale (i.e., saying a letter multiple times if needed) before moving to the next bale. Which of the following, if any, is **NOT** true about this approach?

(a) This approach will always return the correct answer.

(b) This approach is likely to be fast enough for the typical 2 second USACO time limit.

(c) The approach takes $O(N)$ time (time linear in the length of W).

(d) The approach is asymptotically optimal, i.e., any solution to this problem takes $O(N)$ time or worse.

(e) All of the above are true.

2. There is a cleverer (and easier to code) approach that can get the answer based only on...

(a) ...the total number of instances of the letter `A` in W.

(b) ...the maximum number of instances of any one letter in W.

(c) ...the longest consecutive sequence of letters in W that are in alphabetical order.

(d) ...the number of pairs of consecutive letters in W that are not in alphabetical order.

(e) ...the number of pairs of letters (anywhere in W) that are not in alphabetical order.

3. One advantage of using a doubly linked list over an array (e.g., `vector<int>` or `int[]` in C++, `ArrayList<Integer>` or `int[]` in Java, `list` in Python) is that

   (a) Binary searching a linked list is generally much faster than binary searching an array.

   (b) Assuming we have a way to directly access the place in a linked list where we want to insert a new element, inserting into the middle of a linked list is generally much faster than inserting into the middle of an array.

   (c) Assuming we have a way to directly access the place in a linked list where we want to insert a new element, inserting an element at the end of a linked list is generally much faster than inserting an element at the end of an array.

   (d) Linked list operations are generally faster than array operations because linked lists take less total storage space in memory.

   (e) Linked lists can be stored in a single contiguous block of memory, whereas arrays cannot.

4. In a typical associative array (e.g. `map` in C++, `HashMap` in Java, dictionaries in Python), which of the following operations is **NOT** supported in constant (or nearly constant) time?

   (a) Insert a new key-value pair into the associative array.

   (b) Check whether a particular key is in the associative array.

   (c) Check whether a particular value is in the associative array.

   (d) Retrieve the value for a particular key.

   (e) Delete a particular key-value pair from the associative array.

5. It is generally best to avoid using floating point numbers (e.g., `double` in C++ or Java, or the result of using `/` rather than `//` to divide in Python) in coding contest problems where the desired answer is an integer, even when decimals may be involved in the computation. The most important reason for this is that

   (a) Operations on floating point numbers tend to be slower overall.

   (b) All floating point representations of non-integer numbers are inexact.

   (c) Sufficiently large integers cannot be represented exactly as floating point numbers.

   (d) Sufficiently small integers cannot be represented exactly as floating point numbers.

   (e) It is possible that some intermediate values in the computation cannot be represented exactly as floating point numbers, in which case errors can accumulate.

6. **For Questions 6-9, consider the following problem.**

> Some – but not all – of Farmer John's N cows ($2 \leq N \leq 10^5$) are friends with each other. In this problem, friendship is two-way: if a cow A is friends with some other cow B, then cow B is necessarily also friends with cow A.
>
> Two cows are "networked" if there is a path of connections between them formed by friendships. That is, either they are directly friends with each other, or they have a mutual friend, or the first cow is friends with a cow who is friends with a cow who is friends with the second cow, etc.
>
> Farmer John is going to choose two cows who are not already friends and make them friends. Moreover, he will do this in a way that causes the total number of pairs of networked cows to increase as much as possible. These pairs are unordered – i.e., "Bessie and Elsie" does not count as a different pair than "Elsie and Bessie". (Notice that the number will always increase by at least 1, since it includes the pair of cows that Farmer John chose to make friends.)
>
> For example, suppose there are three cows A, B, C, and (only) A and B are already friends. Then if Farmer John makes cows A and C friends, the total number of pairs of networked cows increases from one to three: a total increase of two.

Consider all possible data sets with exactly 10 cows. What is the largest possible increase (in the total number of pairs of networked cows) that Farmer John could achieve?

(a) 9

(b) 18

(c) 25

(d) 36

(e) 45

7. Which of the following high-level approaches would be most appropriate for solving Farmer John's problem efficiently?

(a) Complete search

(b) Binary search

(c) Finding connected components

(d) Sorting

(e) Divide and conquer

8. Which of the following graph traversal techniques would be suitable as a piece of that solution?
   (a) Breadth-first search
   (b) Depth-first search
   (c) Cycle detection
   (d) Either (a) or (b)
   (e) Any of (a), (b), or (c)
9. Consider the following algorithm:
   - For each cow i, find how many friends F_i they have.
   - Sort all the cows (e.g., using MergeSort) in decreasing order of F_i.
   - Find two cows i and j who are not already networked and who have the largest possible F_i + F_j, and make them friends. (If there are multiple such pairs, choose the one with the smallest i, and then if there is still a tie, choose the one with the smallest j.)

   Which of the following statements about this approach is/are true?
   (a) The approach can be implemented in a way that takes $O(N)$ time even in the worst case.
   (b) The approach always returns the correct answer.
   (c) Both (a) and (b) are true.
   (d) Neither (a) nor (b) is true.
   (e) Farmer John is a completely normal person with completely normal hobbies.

10. Here are some statements about connected, undirected graphs with N vertices; assume that $N > 2$. Which of the following statements is **NOT** true?

    (a) A connected, undirected graph that has exactly N vertices and exactly $N - 1$ edges has at least two leaves (i.e., vertices with only a single edge).

    (b) A connected, undirected graph that has exactly N vertices and exactly $N - 1$ edges has exactly one path between any two vertices.

    (c) In a connected, undirected graph that has exactly N vertices and $N - 1$ edges, a vertex might have as few as one edge, or as many as $N - 1$ edges.

    (d) A connected, undirected graph that has exactly N vertices and exactly N edges necessarily has a cycle.

    (e) Removing an edge from a connected, undirected graph that has exactly N vertices and exactly N edges necessarily leaves behind a tree.

11. Suppose that a solution to a problem explicitly checks every possible ordering of N things, and checking each ordering takes linear ($O(N)$) time. Which of these would you expect to be (roughly) the largest value of N for which this approach would be fast enough for the standard USACO 2 second time limit?

    (a) 4

    (b) 8

    (c) 16

    (d) 32

    (e) 64

12. Bessie is in a grid with R rows and C columns; the top left cell is numbered $(1, 1)$, and the bottom right cell is numbered (R, C). Bessie starts in the cell numbered (r, c), and she wants to reach the bottom right cell by only making moves down and/or to the right, without exiting the grid.

    Let $P(x, y)$ denote the number of possible paths Bessie can take from the starting cell to (R, C). We say $P(R, C) = 1$ even though this "path" is trivial.

    Which of the following is **NOT** true?

    (a) $P(x, C) = 1$ for all $1 \leq x \leq R$.

    (b) $P(R, y) = 1$ for all $1 \leq y \leq C$.

    (c) $P(x, y) = P(x + 1, y) + P(x, y + 1)$, for all $x < R$ and $y < C$.

    (d) For any integer value v between 1 and $\max(R, C)$, there is a cell (x, y) for which $P(x, y) = v$.

    (e) $P(1, 1) = R \times C$.

13. Bessie has a directed graph of 10 vertices, in which each vertex has **exactly one** outgoing edge. What is the minimum number of vertices in such a graph that must be part of a cycle – i.e., they can be reached again after leaving them?

    (a) 0

    (b) 2

    (c) 5

    (d) 8

    (e) 10

14. Given a list of N integers (some or all of which might be negative), which of the following problems **cannot** be solved with only a single linear pass through the list and only a constant amount of memory? (Specifically, an amount not proportional to N).

    (a) Find the largest sum of any two different elements anywhere in the list.

    (b) Find the smallest sum of any two different elements anywhere in the list.

    (c) Find the largest difference between any two different elements anywhere in the list.

    (d) Find the smallest difference between any two different elements anywhere in the list.

    (e) Find the smallest product of any two different elements anywhere in the list.

15. Which of the following situations, if any, would **NOT** be suitable for binary search?

    (a) Find the largest natural number N for which $N^{1.01} + 1.01^N \leq 250$.

    (b) Find the largest natural number N that is prime, is made up of nonzero digits, and has its digits sum to 1000.

    (c) Given that your code project was working originally and stopped working at some point (and never started working again), and that your team has submitted a total of 10^{18} changes to the project, find the change that broke the project.

    (d) Given an undirected, connected graph in which each edge is labeled with a number, find the smallest value of N for which we cannot get from a certain vertex to another vertex if we are limited to using only edges labeled N or less.

    (e) All of these situations are good fits for binary search.

16. Given a list L of N integers, Bessie needs to answer N different queries of the form: for indices i and j, with $i < j$, what is the sum of the elements between $L[i]$ and $L[j]$, inclusive?

    What is the best running time that Bessie can achieve, in terms of N?

    (a) $O(N)$

    (b) $O(N \log N)$

    (c) $O(N\sqrt{N})$

    (d) $O(N^2)$

    (e) $O(N^3)$

17. Farmer John wrote a function `maxDepth` that takes the root of a rooted tree and returns the maximum depth of any child of the tree (i.e. the largest number of steps away any child is from the root). If the tree is just a single vertex, `maxDepth` returns 0.

    The villainous Farmer Nhoj stole `maxDepth` and has used it to write `longestPath`, which takes the root of a rooted tree and returns the largest number of steps between any two vertices anywhere in the tree:

    `longestPath`($root$) =

    - If $root$ has no children: 0
    - If $root$ has exactly one child x : `maxDepth`(x) $+ 1$
    - If $root$ has more than one child: maximum, over all pairs x, y of children, of `maxDepth`(x) + `maxDepth`(y) $+ 2$

    Bessie thinks that Farmer Nhoj's `longestPath` algorithm isn't quite correct. She wants to create a tree for which `longestPath` will return the wrong answer. (Bessie gets to decide where the root of this tree is.) What is the smallest possible number of vertices that Bessie's tree could have?

    (a) 5

    (b) 6

    (c) 7

    (d) 8

    (e) Bessie is wrong; Farmer Nhoj's algorithm is actually correct.

18. **For Questions 18-20, consider the following problem.**

> Farmer John has lined up his N cows $(1 \leq N \leq 10^5)$ yet again. Each cow has a name, and it is possible that multiple cows may share the same name.
>
> Farmer John wants to find the length of the longest contiguous sequence of cows in which no two cows have the same name.

Which of the following techniques would be appropriate for solving the problem efficiently?

(a) Sliding window with a fixed window size

(b) Two pointers

(c) Sorting

(d) Binary search

(e) Divide and conquer

19. What is the best running time we can achieve, in terms of N?

(a) $O(N)$

(b) $O(N \log N)$

(c) $O(N\sqrt{N})$

(d) $O(N^2)$

(e) $O(N^3)$

20. Suppose we change the problem to read "...any name in that interval is shared by fewer than K cows in that interval." (Then the original problem is a special case with $K = 2$.)

If we find the best solution we can, by how much does this multiply the running time of our answer to Question 19, as a function of K?

(a) $O(1)$

(b) $O(\log K)$

(c) $O(\sqrt{K})$

(d) $O(K)$

(e) $O(K \log K)$

Don't look at the next page until you've attempted all the problems!

Answers and Explanations

1. **(e)** The solution is correct; it just simulates the process, and there is no reason for Bessie not to say a letter when it will help her make progress. The code is likely to be fast enough; each cycle takes 26 steps, and we will get to say at least one letter per cycle, so we won't need more than $26 \cdot 10^5$ steps. It does take time linear in N, even when the word contains a lot of repeated letters. Even though the solution is not the cleverest, it is asymptotically the best we can do – it is not possible to solve the problem in sub-linear time because we at least have to read the entire input, which takes linear time itself.

   (That said, brute force solutions and direct simulations generally only work at the Bronze level. You may still be able to write some small part of a Silver, Gold, etc. solution this way, but in general you should not count on being able to get away with it!)

2. **(d)** We can observe that Bessie passes through her starting point (right between `Z` and `A`) when (and only when) two adjacent letters in the word are out of alphabetical order, so it suffices to iterate through the word once and count the number of such pairs. This is linear in N, just like our previous solution, so it's no better in big-O terms, but it's much faster (and simpler to code) in practice.

3. **(b)** As long as we have a pointer to the place where we want to insert a new vertex, inserting into a linked list takes constant time: we just need to remove the connections between a vertex and its former follower, and wire up the new vertex as an intermediate between them. Inserting into the middle of an array, however, requires moving all the elements to the right one step forward, since the whole point of an array is that it uses a contiguous block of memory. This lets arrays do something linked lists can't (quickly access, e.g., the hundredth element in the list), but maintaining this property comes at a cost!

   The other choices are wrong because:

   - (a): linked lists don't support fast access to arbitrary elements, which is needed for binary search since we don't know which element we'll need to jump to next.
   - (c): inserting at the end of an array is also fast, since it doesn't require shifting anything around.
   - (d): the opposite is true.
   - (e): arrays are what use a contiguous block of memory. Linked lists can be stored such that all the vertices are in the same region of memory, but they still take more storage space because of the overhead of the extra information about vertices and their connections.

4. **(c)** Associative arrays do not support searching by value. We would potentially have to look at every value in the associative array to find the one we want (or conclude that it is not there).

5. **(e)** This can become particularly nasty when the code requires checking whether some value (which is slightly off from its true value because of accumulated errors) is greater or less than some precise threshold.

   Notice that answer choice (b) is not even a correct statement – numbers that can be expressed as sums of powers of two (including negative powers – e.g., $5.81625 = 4 + 1 + 0.5 + 0.25 + 0.0625$) **can** be represented exactly as floating-point numbers.

6. **(c)** The best-case scenario is when we have two groups of cows that are internally connected, but not connected to each other. Then establishing even one friendship between a cow in one group and a cow in the other group makes all the cows in the first group "networked" with all the cows in the second group, adding a number of new networked pairs equal to the product of the sizes of the two groups.

   This is most effective when the original group sizes are as close as possible, which in this case is 5 and 5, for an answer of 25. (If we had groups of sizes 6 and 4, for instance, then we would get only 24 new networked pairs. This is one example of the "AM-GM inequality", if you're interested in reading more about that...)

7. **(c)** The key idea here is to first identify all the connected components of cows, then choose the largest two and connect them via one pair of cows (it doesn't matter which pair). As suggested by our answer to Question 6, we get the most new networked pairs by picking the largest components we can.

8. **(d)** Either breadth or depth first search work fine for finding connected components, and it comes down to whichever one you prefer implementing. Cycle detection wouldn't really help us here – we only care about which cows can reach which other cows.

9. **(d)** This approach is tempting, but it isn't correct, and it also isn't even fast!

   Consider a case where we have three connected components. Two of them are sets of four cows that are all friends with each other. The other is a long cycle of 1000 cows, each of which is only friends with its two neighbors in the cycle.

   Then this algorithm will connect a cow from one of the four-cow components to another cow in one of the four-cow components, since each of those cows has 3 friends, as opposed to 2 friends for each of the cows in the large cycle. But we could have gotten many more networked pairs by connecting a cow from a four-cow component to a cow from the 1000-cow component.

   The approach also isn't fast, because it's not straightforward how to identify a pair with the largest possible $F_i + F_j$ score that isn't already networked. It could involve comparing a large number of pairs of cows, pushing the running time to $O(N^2)$. As an extreme case, suppose that all pairs of cows are already friends, except for one pair. That pair will be the last one that this algorithm checks!

10. **(e)** This statement is only true if the edge is removed from the graph's cycle. Removing an edge might disconnect the graph, leaving behind two separate connected components instead of a single tree.

11. **(b)** The number of orderings of N things is $N!$, N factorial, which is $(N)(N-1)...(1)$. It's good to have a sense of how fast this function blows up. $8!$ is only 40320, but $16!$ is 20922789888000... way too many times for a computer to repeat anything in two seconds! (Check back in a hundred years, though...)

12. **(e)** The other answer choices correctly describe a recurrence that gives us the answer starting from any cell. For example, for $R = 4, C = 6$, we have:

56	35	20	10	4	1
21	15	10	6	3	1
6	5	4	3	2	1
1	1	1	1	1	1

Notice that we can fill in the entire table by beginning with the last row and column, then filling in the next-to-last row and next-to-last column, and so on.

But $P(1,1)$ is generally much larger than $R \cdot C$, since there is a compounding effect as we look up and to the left in the grid.

13. **(b)** At first it might seem that every vertex has to be part of a cycle, but it is possible that some vertices feed directly into (or are parts of chains feeding into) cycles, and are not in cycles themselves.

    Consider a case where vertex 1 points to vertex 2, which points back to vertex 1, and all of vertices 3 through 10 point to vertex 1. Or vertex 3 points to vertex 4, and so on, with vertex 9 pointing to vertex 10, which points to vertex 1.

    An answer of 0 is not possible because there must be a cycle somewhere in the graph. Every vertex has to point somewhere, and the vertex it points to has to point somewhere, and so on... eventually that "somewhere" must be one of the vertices we have already seen.

    This situation (with vertices or chains of vertices pointing into cycles) has come up multiple times in USACO Silver problems.

14. **(d)** We can solve this by eliminating the other choices: (a) is the sum of the largest two elements in the list, (b) is the sum of the two smallest, and (c) is the largest minus the smallest. It's easy to step through the list while keeping track of the two largest (or two smallest) values that we've seen so far.

    (e) is a little trickier, but it uses the same quantities we have already discussed: If all the numbers are nonnegative, it is the product of the smallest two. Otherwise, it is the product of the most negative number and the most positive number.

    But there is no similar algorithm for the smallest difference. The easiest way to find this is to sort the list and then compare all consecutive pairs of numbers, and take the smallest difference. But if we can't sort the list, and we can't keep track of enough memory to effectively sort the list as we go, there is no small enough set of information that we can keep track of.

    As a thought experiment: suppose we have a list where the first elements are $10, 20, 40, 80, 160$... and so on, and we look at all of them except for the last. But then the last element could be closest to any of these numbers – how could we remember enough information without remembering a linear amount of it?

15. **(b)** Binary search works when a property holds for all numbers below a certain threshold and doesn't hold for all numbers above a certain threshold. (Or vice versa, swapping "below" and "above".)

    The situations in (a), (c), and (d) have this property, but the situation in (b) does not. The "good" numbers that have both of the desired properties are spaced out in an irregular way, with many "bad" numbers between them. Getting a verdict for one number doesn't tell us whether to look

higher or lower. The best way to solve this would probably be some kind of brute force that investigate all 1000-digit candidates (of which there is only one), then all 999-digit candidates (in order of decreasing size), etc., until an answer is found.

16. **(a)** It would be inefficient to repeatedly compute the sum of the elements between each query's two indices, because we'd end up computing the same information multiple times.

    But we can use the idea of prefix sums to avoid that! Before we process any queries, we can create an array with the cumulative sums of all the elements up to each point. That is, for the array $[1, 3, -2, 5, 0, -4]$, we'd get $[1, 4, 2, 7, 7, 3]$. Then to solve a query like $(3, 6)$, for example, we would look up the cumulative sum up to the sixth index (the first through sixth elements) and subtract off the cumulative sum up the second index (the first and second elements), and the remainder would be exactly what we want: the sum of the third through sixth elements.

    It takes $O(N)$ time to make this array, and then each of the N queries takes $O(1)$ time, so the overall running time is only $O(N)$.

17. **(b)** The problem with Farmer Nhoj's approach is that it implicitly assumes that the longest path in the tree must go through the root vertex. But this isn't necessarily the case! Consider a tree like this

    ```
    V
    |
    W
    |
    X-R
    |
    Y
    |
    Z
    ```

    where vertex `R` is the root. Then the longest path (`V-W-X-Y-Z`) is of length 4 and does not go through the root at all. But the algorithm incorrectly returns 3, since it adds 1 to `maxDepth(X)`, and `maxDepth(X)` is 2.

    We can check (with some experimentation) that this is the smallest example that breaks Nhoj's algorithm.

18. **(b)** Our best approach is to use two pointers – a "left" pointer and a "right" pointer. They both begin at the beginning of the list, and then we advance the right pointer forward, keeping track of which names we have seen using, e.g., a set structure. When the right pointer encounters a name that is already in the set, we:

    - Take the difference of the indexes of the right and left pointers – this is our current best guess at the length of the longest interval without a repeated name.
    - Move the left pointer forward until it is just past the first instance of the repeated name. Each time we move the left pointer past a name, we delete that name from the set.
    - Add the name pointed to by the right pointer back to the set.

Then we resume moving the right pointer ahead until we encounter another repeated name, and so on. At the end of the algorithm, the longest interval we ever found is our answer.

19. **(a)** In the description in our answer to Question 18, we see that both pointers only ever move forward, so they collectively move only a linear number of times. As long as managing the set also takes linear time (constant time per check), the whole algorithm is linear.

20. **(a)** Surprisingly, this doesn't make the algorithm much more complex! We switch out our set for a map, and then store the number of times we've seen each name, instead of just *whether* we've seen it or not.