Q6

Consider the following algorithm for calculating the nth Fibonacci number: function fib(n): if n <= 1 return n else return fib(n-1) + fib(n-2). What is the time complexity of this algorithm?

A O(n)

B O(log n)

C O(n^2)

[✓ D O(2^n)]

Hide Answer

Report

Q7

An algorithm supposed to calculate the sum of numbers from 1 to n returns a higher value than expected. What is the most likely mistake?

A Starting the loop from 0

B Not initializing the sum variable

[✓ C Adding n twice]

D All of the above

Hide Answer

Report

Answer: Adding n twice

Explanation

If the algorithm adds the final number n twice, it would return a higher value than expected. Ensuring that each number is added exactly once is crucial for correct sum calculation.

Q8

Given an algorithm that always returns the first element in a sorted list instead of the smallest, what is likely the issue?

[✓ A The algorithm incorrectly assumes the first element is the smallest]

B The list is not properly sorted

C A loop iterates incorrectly

D All of the above

Hide Answer

Report

Answer: The algorithm incorrectly assumes the first element is the smallest

Explanation

In a sorted list, the smallest element is indeed the first one, but if the algorithm's purpose is to find the smallest element in any list, assuming the first element is always the smallest without sorting or comparison is incorrect.

Q13 Which of the following best describes the time complexity of inserting an element into a binary search tree?

A O(1)

[✓ B O(log n)]

C O(n)

D O(n log n)

Hide Answer

Report

Answer: O(log n)

Explanation

The average case time complexity for inserting an element into a binary search tree is O(log n), assuming the tree is balanced.

Q14 What is the worst-case time complexity of quicksort?

(A) O(n log n)	(B) O(n)
(C) O(n^2)	(D) O(log n)

Hide Answer ⓘ Report

Answer: O(n^2)

Explanation

The worst-case time complexity of quicksort is O(n^2), which occurs when the pivot selection is poor, such as selecting the smallest or largest element as the pivot in every partition step.

Q15 How does the space complexity of an iterative solution compare to a recursive solution for the same problem?

(A) Iterative solutions always use more space	(B) Recursive solutions always use more space
(C) Depends on the specific problem	(D) They use the same amount of space

Hide Answer ⓘ Report

Answer: Depends on the specific problem

Explanation

The space complexity comparison between iterative and recursive solutions depends on the specific problem and implementation details. In some cases, recursive solutions may use more stack space, while iterative solutions might use more heap space or vice versa.

Q18 Analyze the time complexity of the following function: def func(n):
if n <= 1:
return else func(n/2) + func(n/2)

- (A) O(n)
- (B) O(log n)
- (C) O(n^2)
- (D) O(n log n)

Hide Answer

ⓘ Report

Answer: O(n log n)

Explanation

This function makes two calls to itself with half of n, leading to a time complexity of O(n log n) due to the division of n and the depth of recursion.

Q19 An algorithm that should run in O(n log n) time complexity runs significantly slower.
The likely cause is:

- (A) Incorrect base case in recursion
- (B) Excessive memory allocation
- (C) Poor choice of pivot in sorting
- (D) All of the above

Hide Answer

ⓘ Report

Answer: Poor choice of pivot in sorting

Explanation

A poor choice of pivot in sorting algorithms like quicksort can degrade performance to O(n^2), significantly slower than the expected O(n log n).

Q21 A recursive algorithm expected to have a time complexity of O(log n) is running slower.
The likely issue is:

- (A) Not halving the input on each recursive call
- (B) Incorrect termination condition
- (C) Stack overflow
- (D) All of the above

Hide Answer ⓘ Report

Answer: Not halving the input on each recursive call

Explanation

If the recursive algorithm does not halve the input on each call, it will not achieve the expected O(log n) time complexity, resulting in slower performance.

Q24 Which of the following is NOT a valid reason to use a string builder in Java instead of concatenating strings using the + operator?

- (A) It reduces memory usage
- (B) It is faster for concatenating multiple strings
- (C) It is immutable
- (D) It can be used in multi-threaded environments

Hide Answer ⓘ Report

Answer: It is immutable

Explanation

A string builder is mutable, which is why it's preferred over string concatenation with the + operator in scenarios involving multiple concatenations. This mutability leads to less memory usage and faster operations since it doesn't create a new object for each concatenation.

Q25 In an unsorted array of integers, what is the best time complexity achievable for searching for a specific value?

A O(1)

B O(n)

C O(log n)

D O(n^2)

Hide Answer

Report

Answer: O(n)

Explanation

In an unsorted array, the best time complexity achievable for searching for a specific value is O(n), as it might be necessary to check each element until the desired value is found.

Q26 Considering a character array representing a string, what is the space complexity for storing this string?

A O(1)

B O(n)

C O(log n)

D O(n^2)

Hide Answer

Report

Answer: O(n)

Explanation

The space complexity for storing a string in a character array is O(n), where n is the number of characters in the string, as each character requires a fixed amount of space.

Q27 Which operation has a worse time complexity in a dynamic array when it needs to expand its size?

A Accessing an element by index	B Appending an element at the end
🟩 C Inserting an element at the beginning	🟥 D Searching for an element

Hide Answer ⓘ Report

Answer: Inserting an element at the beginning

Explanation

Inserting an element at the beginning of a dynamic array when it needs to expand its size involves shifting all existing elements, making it a costly operation. Appending is generally efficient due to amortized constant time, but expansion requires copying elements to a new array, which is less frequent.

Q28 What does the following Python code snippet return?
arr = ['a', 'b', 'c', 'd'];
print(arr[1:3])

Ⓐ ['a', 'b']

Ⓑ ['b', 'c']

Ⓒ ['c', 'd']

Ⓓ ['b', 'c', 'd']

Hide Answer

ⓘ Report

Answer: ['b', 'c']

Explanation

The Python code snippet returns ['b', 'c'], as slicing in Python is inclusive of the start index and exclusive of the end index.

Q29 Given an array of integers, which of the following operations will NOT mutate the original array in JavaScript?

Ⓐ arr.sort()

Ⓑ arr.push(5)

Ⓒ [...arr, 5]

Ⓓ arr.pop()

Hide Answer

ⓘ Report

Answer: [...arr, 5]

Explanation

The operation [...arr, 5] creates a new array with the elements of arr followed by 5 and does not mutate the original array. The other operations (sort(), push(5), and pop()) modify the original array.

Q30 What is the result of concatenating two arrays in Python using the + operator, arr1 = [1, 2, 3] and arr2 = [4, 5, 6]?

- A A new array [1, 2, 3, 4, 5, 6]
- B The original arrays are mutated to include the elements of the other
- C A syntax error
- D None of the above

Hide Answer

Report

Answer: A new array [1, 2, 3, 4, 5, 6]

Explanation

Concatenating two arrays in Python with the + operator results in a new array containing the elements of both arrays in the order they appear. The original arrays are not mutated by this operation.

Q32 Why does string.split('').reverse().join('') in JavaScript return a reversed string?

- A The split method incorrectly splits the string
- B The reverse method does not work on strings
- C The join method concatenates incorrectly
- D None of the above

Hide Answer

Report

Answer: None of the above

Explanation

The split('') method splits the string into an array of characters, reverse() reverses the array in place, and join('') concatenates the reversed array back into a string. This sequence of operations correctly reverses the string.

Q33 In a program designed to find the longest string in an array of strings, the output is always the first string. What is the likely error?

- A Not updating the longest string variable inside the loop
- B Using the wrong comparison operator
- C Not initializing the longest string variable
- D All of the above

Hide Answer

Report

Answer: Not updating the longest string variable inside the loop

Explanation

If the program always outputs the first string as the longest, it likely means the variable intended to hold the longest string is not being updated correctly inside the loop that iterates through the array to compare string lengths.

Q38 How do you detect a cycle in a linked list?

- A By checking if the next pointer of any node is null
- B Using a hash table to store visited nodes
- C Comparing each node with every other node
- D Using two pointers at different speeds

Hide Answer

Report

Answer: Using two pointers at different speeds

Explanation

The technique of using two pointers (often called slow and fast pointers, where the fast pointer moves twice as fast as the slow pointer) can detect cycles. If the two pointers meet, a cycle exists; if the fast pointer reaches the end of the list, the list does not have a cycle.

Q40 What does the following code snippet do?
node.next = node.next.next;

- (A) Deletes the next node in the list
- (B) Inserts a new node after the current one
- (C) Swaps two nodes
- (D) Duplicates the next node

Hide Answer ⓘ Report

Answer: Deletes the next node in the list

Explanation

This code snippet deletes the next node in a singly linked list by bypassing it. It sets the current node's next pointer to point to the node after the next, effectively removing the next node from the list.

Q41 Consider a linked list implementation.
What does head = newNode;
newNode.next = oldHead;
accomplish?

- (A) Reverses the linked list
- (B) Adds a new node to the end of the list
- (C) Adds a new node at the beginning of the list
- (D) Deletes the head node

Hide Answer ⓘ Report

Answer: Adds a new node at the beginning of the list

Explanation

This code snippet adds a new node at the beginning of the linked list. It assigns the new node to be the head of the list and links the new node to the previous head, effectively inserting it at the start.

Q44 Why might a find function in a linked list return null for existing values?

- (A) The comparison logic is incorrect
- (B) It starts at the wrong node
- (C) It skips over nodes
- (D) Any of the above

Hide Answer ⓘ Report

Q42

In a doubly linked list, each node has a value, a prev, and a next pointer.
How do you insert a new node after a given node?

A Update givenNode.next and newNode.prev

B Update givenNode.next, newNode.next, newNode.prev, and the next node's prev

C Only update newNode.next

D Only update newNode.prev

Hide Answer

Report

Answer: Update givenNode.next, newNode.next, newNode.prev, and the next node's prev

Explanation

To insert a new node after a given node in a doubly linked list, you need to update the next pointer of the given node, the prev and next pointers of the new node, and the prev pointer of the node that comes after the new node to maintain the bidirectional linkage.

Q45

In a function intended to add a node at a specific index in a linked list, the node is added at the end regardless of the index.
What's the error?

A Not iterating through the list correctly

B Not checking if the index is within bounds

C Both A and B

D None of the above

Hide Answer

Report

Answer: Both A and B

Explanation

The error likely lies in not correctly iterating through the list to reach the specified index and not checking if the index is within the bounds of the list. This results in the node being added at the end by default, as the conditions to insert it at the correct position are not met.

Q55 A queue implementation returns incorrect elements when dequeuing.
What could be the problem?

- (A) The enqueue operation places elements at the front
- (B) The dequeue operation removes elements from the wrong end
- (C) Both A and B
- (D) None of the above

Hide Answer

(!) Report

Answer: The dequeue operation removes elements from the wrong end

Explanation

If a queue implementation returns incorrect elements when dequeuing, it's likely that the dequeue operation is incorrectly removing elements from the wrong end, not following the FIFO (First In, First Out) principle.

Q57 A stack implemented using an array throws an index out of bounds exception.
What is the most probable cause?

- (A) Incorrectly initializing the stack size
- (B) Exceeding the stack capacity without resizing
- (C) Incorrect index calculation for push/pop
- (D) All of the above

Hide Answer

(!) Report

Q61 Which traversal method is used to visit nodes in a level-by-level manner from left to right in a tree?

- (A) Preorder
- (B) Inorder
- (C) Postorder
- (D) Level-order

Hide Answer

(!) Report

Answer: Level-order

Explanation

Level-order traversal visits nodes level by level from left to right, which is especially useful for trees to understand their structure layer by layer.

Q62 What data structure is best suited for implementing a graph's adjacency list?

- Ⓐ Array
- [✓ Ⓑ Linked list]
- Ⓒ Hash table
- Ⓓ Tree

Hide Answer ⓘ Report

Answer: Linked list

Explanation

A linked list is best suited for implementing a graph's adjacency list because it efficiently manages the dynamic sizes of adjacency lists, allowing for easy addition and removal of edges.

Q64 What is the property of a balanced binary search tree (BST)?

- [✓ Ⓐ The left and right subtrees' heights differ by at most one]
- Ⓑ Each subtree is a full tree
- Ⓒ Each subtree is a complete binary tree
- Ⓓ All leaf nodes are at the same level

Hide Answer ⓘ Report

Q67 In graph theory, how is a weighted edge represented in an adjacency list?

- Ⓐ As a list of vertex pairs
- Ⓑ As a list of vertices with associated edge lists
- [✓ Ⓒ As a list of tuples, each containing a vertex and the edge weight]
- [✗ Ⓓ As a two-dimensional matrix]

Hide Answer ⓘ Report

Answer: As a list of tuples, each containing a vertex and the edge weight

Explanation

In an adjacency list, a weighted edge is typically represented as a list of tuples, where each tuple contains a vertex and the weight of the edge connecting to that vertex, allowing the graph to store the cost or length of each connection.

Q68 What algorithm can be used to detect a cycle in a directed graph?

- (A) Depth-first search (DFS)
- (B) Breadth-first search (BFS)
- (C) Kruskal's algorithm
- (D) Dijkstra's algorithm

Hide Answer

(!) Report

Answer: Depth-first search (DFS)

Explanation

Depth-first search (DFS) can be used to detect cycles in directed graphs by tracking visited nodes and checking for back edges, which indicate a cycle when a node is encountered again in the same path.

Q69 You implemented a tree but notice that child nodes are not correctly associated with their parents.
What might be the issue?

- (A) The tree is incorrectly initialized as a graph
- (B) Child nodes are added to the wrong parent
- (C) The tree structure does not support hierarchy
- (D) Nodes are not properly linked

Hide Answer

(!) Report

Q70 A graph's adjacency matrix does not reflect the correct connections between nodes.
What is a possible mistake?

- (A) The matrix dimensions are incorrect
- (B) Edges are added to the wrong cells in the matrix
- (C) The matrix is not updated when edges are added or removed
- (D) Both B and C

Hide Answer

(!) Report

Answer: Edges are added to the wrong cells in the matrix

Explanation

If a graph's adjacency matrix does not reflect the correct connections, a possible mistake is that edges are being added to the wrong cells, indicating a mismatch between the vertices and their corresponding matrix indices.

Q73 Which sorting algorithm is inherently stable?

- A QuickSort
- B HeapSort
- C MergeSort
- D BubbleSort

Hide Answer

Report

Answer: MergeSort

Explanation

MergeSort is stable, maintaining the original order of equal elements, making it suitable for sorting complex records.

Q76 Which sorting algorithm is most efficient for a dataset with a known, limited range of integer values?

- A QuickSort
- B BubbleSort
- C CountingSort
- D InsertionSort

Hide Answer

Report

Answer: CountingSort

Explanation

CountingSort is ideal for datasets with a limited range of integers, as it counts occurrences rather than performing direct comparisons, leading to efficient sorting.

Q78 Why is QuickSort generally preferred over MergeSort for sorting arrays in practice?

- A Lower space complexity
- B Faster average sorting times
- C Simpler to implement
- D Stable sorting guaranteed

Hide Answer

Report

Q80 What technique improves QuickSort's performance on small arrays?

- A Insertion sort hybrid
- B Random pivot selection
- C Decreasing recursion depth
- D Increasing stack size

Hide Answer

Report

Answer: Insertion sort hybrid

Explanation

For small arrays, QuickSort's performance is improved by switching to Insertion Sort, a simpler sorting algorithm that is more efficient for small data sets due to its lower overhead.

Q85 QuickSort is causing a stack overflow error.
What's a probable cause?

- A Excessive recursion on large datasets
- B Incorrect pivot selection leading to unbalanced partitions
- C Non-terminating recursive calls
- D All conditions are met but still failing

Hide Answer

Report

Answer: Incorrect pivot selection leading to unbalanced partitions

Explanation

Incorrect pivot selection in QuickSort can lead to highly unbalanced partitions, causing excessive recursion depth on nearly sorted or certain patterned datasets, which may result in a stack overflow error.

Q86 What is a hash table?

- A A data structure that stores key-value pairs in a linear array
- B A data structure that organizes data for quick search, insertion, and deletion based on keys
- C A data structure for storing hierarchical data
- D A data structure that stores data in nodes with a key and multiple values

Hide Answer

Report

Q87 Which of the following is a common use case for a hash table?

(A) Implementing a database indexing system

(B) Storing preferences of a user in a web application

(C) Performing quick searches in a large dataset

(D) All of the above

Q89 What is the primary challenge in designing a hash function for a hash table?

(A) Ensuring it is reversible

(B) Minimizing the occurrence of collisions

(C) Ensuring it produces a unique output for each input

(D) Maximizing the computational complexity

Hide Answer ⓘ Report

Answer: Minimizing the occurrence of collisions

Explanation

The primary challenge in designing a hash function is minimizing the occurrence of collisions. A good hash function distributes keys uniformly across the hash table, reducing the likelihood of collisions and thus maintaining efficient access times.

Q90 Which technique is commonly used to resolve collisions in a hash table?

(A) Linear probing

(B) Using a binary search tree

(C) Doubling the size of the table when full

(D) Storing all entries in a single linked list

Hide Answer ⓘ Report

Answer: Linear probing

Explanation

Linear probing is a common collision resolution technique where, upon encountering a collision, the hash table searches for the next available slot by moving sequentially through the table. This method is simple to implement and can be effective in distributing entries evenly across the table.

Q91 In the context of hash tables, what does "load factor" refer to?

[✓ Ⓐ The ratio of the number of entries to the number of buckets in the table]

Ⓑ The maximum number of collisions allowed before resizing

Ⓒ The percentage of keys that are null

Ⓓ The average search time for an entry

Q92 What strategy can significantly reduce the chance of collisions in a hash table?

[✓ Ⓐ Using a prime number for the size of the hash table]

Ⓑ Increasing the size of the keys

Ⓒ Decreasing the number of entries

Ⓓ Using multiple hash functions for the same key

Q93 How do you access a value stored in a hash table given its key?

Ⓐ By computing the hash of the key and searching linearly

[✗ Ⓑ By directly indexing the array with the key]

[✓ Ⓒ By computing the hash of the key and using it as an index]

Ⓓ By sorting the keys and performing a binary search

Hide Answer ⓘ Report

Answer: By computing the hash of the key and using it as an index

Explanation

To access a value in a hash table, you compute the hash of the key, which gives an index in the array or bucket array where the value can be found. This operation is typically O(1), making hash tables highly efficient for data retrieval.

Q97 A developer notices that retrieval times from a hash table are consistently slow.
What is a likely reason?

Ⓐ The hash function is too complex

[✓ Ⓑ The load factor is too high, causing excessive collisions]

Ⓒ The keys are not distributed uniformly

Ⓓ All entries are stored in a single bucket

Hide Answer ⓘ Report

Q94 What is the most common method to handle collisions in a hash table programmatically?

- A Open addressing with linear probing
- B Storing values in a list at each index
- C Doubling the hash table size on a collision
- D Using a secondary hash function

Hide Answer ⓘ Report

Answer: Open addressing with linear probing

Explanation

Open addressing with linear probing is a common method to programmatically handle collisions in a hash table. It involves finding the next available slot within the table array by moving sequentially from the point of collision, thereby resolving the collision without needing additional data structures.

Q95 How do you ensure that a hash table remains efficient as more entries are added?

- A By periodically decreasing the table size
- B By rehashing all entries into a larger table when the load factor reaches a threshold
- C By limiting the number of entries
- D By converting the table to a binary search tree on overflow

Hide Answer ⓘ Report

Answer: By rehashing all entries into a larger table when the load factor reaches a threshold

Explanation

Ensuring efficiency as entries are added involves rehashing all entries into a larger table when the load factor reaches a certain threshold. This process, known as resizing or rehashing, helps maintain a low load factor, reducing the likelihood of collisions and keeping access times short.

Q96 Which approach is best for storing values that have the same hash key in a hash table?

Ⓐ Overwriting the previous value

Ⓑ Linking new values to the existing ones in a linked list

Ⓒ Ignoring new values with duplicate keys

Ⓓ Storing values in an adjacent table

Hide Answer ⓘ Report

Answer: Linking new values to the existing ones in a linked list

Explanation

Linking new values to the existing ones in a linked list, known as chaining, is an effective approach for handling collisions caused by multiple keys having the same hash. This method allows multiple values to coexist at the same index by extending the collision resolution beyond a single slot.

Q98 During testing, a hash table's add operation sometimes fails to insert new elements. What could be the problem?

Ⓐ The hash function always returns the same value

Ⓑ Collisions are not handled correctly

Ⓒ The table is full and cannot resize

Ⓓ The key is null

Hide Answer ⓘ Report

Answer: Collisions are not handled correctly

Explanation

If adding new elements to a hash table sometimes fails, it may indicate that collisions are not being handled correctly. Effective collision resolution strategies, such as open addressing or chaining, are necessary to ensure that all elements can be inserted even when hashes collide.

Q102 What distinguishes a greedy algorithm from a dynamic programming approach?

Ⓐ Greedy algorithms consider all possible solutions before making a choice

Ⓑ Dynamic programming uses recursion to solve subproblems

Ⓒ Greedy algorithms make the locally optimal choice at each step

Ⓓ Dynamic programming cannot handle overlapping subproblems

Hide Answer ⓘ Report

Q99 A hash table implementation experiences intermittent performance degradation.
What might be causing this issue?

- (A) Inconsistent hash function performance
- (B) Varying sizes of entries
- (C) Periodic table resizing operations
- (D) Non-uniform key distribution

Hide Answer

(!) Report

Answer: Periodic table resizing operations

Explanation

Intermittent performance degradation in a hash table could be caused by periodic table resizing operations. Resizing, especially rehashing all entries into a larger table, can be computationally expensive and may temporarily affect performance, particularly if triggered frequently as entries are added.

Q101 In which scenario would a greedy algorithm be preferred over dynamic programming?

- (A) When an optimal solution needs to be guaranteed for all cases
- (B) When subproblems overlap and are dependent
- (C) When subproblems are independent and a local optimum is acceptable
- (D) When the problem size is very small

Hide Answer

(!) Report

Answer: When subproblems are independent and a local optimum is acceptable

Explanation

Greedy algorithms are preferred when subproblems are independent, and finding a local optimum at each step is acceptable for finding a solution. Unlike dynamic programming, greedy algorithms make a locally optimal choice at each step with the hope of finding a global optimum, which is not always guaranteed.

Q141 What distinguishes dynamic programming from the divide and conquer approach?

- (A) Dynamic programming requires that the problem has overlapping subproblems, whereas divide and conquer does not
- (B) Dynamic programming uses only recursion, while divide and conquer does not
- (C) Dynamic programming is used only for optimization problems
- (D) Divide and conquer algorithms are not applicable to problems with optimal substructure

Q108 What technique is used in dynamic programming to transform a recursive solution into an iterative one?

Ⓐ Memoization

Ⓑ Tabulation

Ⓒ Backtracking

Ⓓ Divide and conquer

Hide Answer

ⓘ Report

Answer: Tabulation

Explanation

Tabulation, also known as the bottom-up approach, is used in dynamic programming to transform a recursive solution into an iterative one. It involves filling up a table (usually an array) iteratively and solving the problem by first solving all related subproblems. This approach starts with the simplest subproblems and uses their solutions to build up solutions to more complex subproblems.

Q110 What is a key advantage of using dynamic programming over naive recursion for problems like calculating the nth Fibonacci number?

Ⓐ It reduces the computational complexity

Ⓑ It eliminates the need for calculation

Ⓒ It uses less memory

Ⓓ It relies on simpler mathematical concepts

Hide Answer

ⓘ Report

Q112 A dynamic programming solution is running slower than expected.
What could be a reason?

Ⓐ The problem does not have overlapping subproblems

Ⓑ Subproblems are not being correctly memoized

Ⓒ There are too many subproblems

Ⓓ The base cases are defined incorrectly

Q113 What issue could arise when implementing a greedy algorithm for a complex optimization problem?

Ⓐ Overlooking better solutions due to making premature decisions

Ⓑ Incorrectly assuming the problem has overlapping subproblems

Ⓒ Using too much memory

Ⓓ Not using recursion enough

Q117 What does the Bellman-Ford algorithm accomplish?

[✓ A Finding the shortest path in a graph with negative edge weights]

B Creating a minimum spanning tree

C Finding the maximum flow in a network

D Detecting and breaking cycles in a directed graph

Q118 What is the primary difference between Prim's and Kruskal's algorithms?

A Prim's algorithm is used for shortest path finding, while Kruskal's is used for minimum spanning trees

[✓ B Prim's requires a starting vertex; Kruskal's does not]

C Prim's is a greedy algorithm; Kruskal's is not

[✗ D Prim's can handle negative edge weights; Kruskal's cannot]

Hide Answer ⓘ Report

Answer: Prim's requires a starting vertex; Kruskal's does not

Explanation

Prim's and Kruskal's algorithms both find a minimum spanning tree for a weighted undirected graph. The primary difference is that Prim's algorithm requires a starting vertex and grows the MST one vertex at a time, while Kruskal's algorithm does not require a starting vertex and adds edges in order of increasing weight, ensuring no cycles are formed.

Q119 Why are topological sorts important in graph algorithms?

A They are used to detect cycles in undirected graphs

[✓ B They provide a way to schedule tasks with dependencies]

C They find the shortest path in weighted graphs

D They compute the maximum flow in networks

Q121 Which algorithm is used to find the strongly connected components in a directed graph?

A Dijkstra's algorithm

B Bellman-Ford algorithm

[✓ C Kosaraju's algorithm]

D Floyd-Warshall algorithm

Q120 How do you implement a graph traversal to check if a graph is bipartite?

A By using a depth-first search and assigning colors to each node

B By finding the shortest path between all pairs of nodes

C By creating a minimum spanning tree

D By performing a matrix multiplication

Hide Answer

Report

Answer: By using a depth-first search and assigning colors to each node

Explanation

To check if a graph is bipartite, you can use either depth-first search (DFS) or breadth-first search (BFS) to assign colors (e.g., two colors) to each node, alternating colors as you traverse the graph. If you can successfully color the graph this way without conflict, the graph is bipartite; otherwise, it is not.

Q122 How is the all-pairs shortest path problem solved in a graph with no negative cycles?

A Using Dijkstra's algorithm repeatedly for each vertex

B Using the Bellman-Ford algorithm repeatedly for each vertex

C Using Floyd-Warshall algorithm

D Using Prim's algorithm

Hide Answer

Report

Answer: Using Floyd-Warshall algorithm

Explanation

The Floyd-Warshall algorithm is ideal for solving the all-pairs shortest path problem in graphs, including those with negative weights but no negative cycles. It systematically compares all paths through the graph to find the shortest paths between all pairs of vertices, utilizing a dynamic programming approach.

Q123 In a graph, how do you determine whether adding an edge would create a cycle?

A By performing a topological sort

B By checking if the edge connects vertices in the same strongly connected component

C By using a union-find data structure

D By calculating the graph's diameter

Hide Answer

Report

Q124 Why might a breadth-first search (BFS) algorithm fail to find the shortest path in a weighted graph?

- A Because BFS does not account for edge weights
- B Because BFS only works on unweighted graphs
- C Because the graph is not properly connected
- D Because the starting node is chosen incorrectly

Hide Answer ⓘ Report

Answer: Because BFS does not account for edge weights

Explanation

BFS is designed for unweighted graphs where all edges are considered equal. In a weighted graph, BFS might fail to find the shortest path because it does not consider the weight of the edges, only the number of edges. For weighted graphs, algorithms like Dijkstra's are more suitable for finding the shortest path.

Q126 What could cause Floyd-Warshall algorithm to give incorrect results for shortest paths?

- A Failing to initialize the distance matrix correctly
- B Not iterating through all vertex pairs
- C Incorrectly handling negative cycles
- D All of the above

Hide Answer ⓘ Report

Answer: Failing to initialize the distance matrix correctly

Explanation

Incorrect initialization of the distance matrix in the Floyd-Warshall algorithm can lead to wrong results. The matrix must correctly represent the distances between all pairs of vertices at the start, including setting the distance from a vertex to itself as zero and considering direct edge weights between vertices. Any mistakes in this setup can affect the entire computation.

Q127 What characteristic defines a binary heap?

- A A binary tree that is completely filled, except possibly for the bottom level, which is filled from left to right
- B A binary tree where each node has a value greater than or equal to its children
- C A binary tree where each node has a value less than or equal to its children
- D Both A and B

Q128 In the context of tries, what does a node represent?

- Ⓐ A letter in a string
- Ⓑ A complete string
- Ⓒ A pointer to another trie
- Ⓓ A and C

Hide Answer

ⓘ Report

Answer: A letter in a string

Explanation

In a trie, which is a type of search tree, each node represents a letter in a string. Tries are used to store a dynamic set or associative array where the keys are usually strings. Unlike binary search trees, no node in the trie stores the key associated with that node; instead, its position in the trie defines the key with which it is associated.

Q129 What is a key advantage of using a Fibonacci heap over a binary heap?

- Ⓐ Faster merge operation
- Ⓑ Better space complexity
- Ⓒ Fixed time insertion
- Ⓓ A and C

Hide Answer

ⓘ Report

Q131 How does a trie differ from a hash table when it comes to storing strings?

- Ⓐ Tries do not require hash functions and can provide alphabetical ordering
- Ⓑ Hash tables are faster for insertion and deletion
- Ⓒ Tries take up less space
- Ⓓ A and C

Hide Answer

ⓘ Report

Q132 What is the significance of amortized analysis in the context of advanced data structures like splay trees or Fibonacci heaps?

- A It provides the worst-case time complexity for any single operation
- [✓ B It shows the average time complexity over a sequence of operations]
- C It guarantees constant time complexity for all operations
- D It reduces the space complexity of the data structure

Hide Answer

Report

Answer: It shows the average time complexity over a sequence of operations

Explanation

Amortized analysis is significant for understanding the efficiency of advanced data structures like splay trees or Fibonacci heaps because it provides a way to calculate the average time complexity over a sequence of operations. This is particularly useful for these data structures, where certain operations might be costly, but the average cost of operations remains low when considered over a series of operations.

Q133 How do you insert a new key into a trie?

- A Create a new node for every character of the key and link them
- [✓ B Reuse existing nodes for the key if they match and create new nodes only when necessary]
- C Insert the key at the root
- D B and C

Hide Answer

Report

Answer: Reuse existing nodes for the key if they match and create new nodes only when necessary

Explanation

Inserting a new key into a trie involves starting at the root and reusing existing nodes for each character of the key if they match, creating new nodes only when necessary. This process continues until all characters of the key have been inserted, making tries an efficient data structure for storing sets of strings or dictionaries.

Q134 What operation is typically more complex to implement in a balanced binary search tree compared to a binary heap?

- A Finding the maximum value
- B Insertion
- C Deletion
- D Finding the minimum value

Hide Answer

Report

Answer: Deletion

Explanation

Deletion is typically more complex to implement in a balanced binary search tree compared to a binary heap. While finding the minimum or maximum value and insertion can be managed with relative ease in both structures, deletion in a balanced BST requires careful handling to maintain the tree's balanced property, often necessitating additional rotations or adjustments.

Q135 In a min heap, how do you ensure that the structure remains valid after inserting a new element?

- A By swapping the new element with the root if it's smaller
- B By placing the new element in the leftmost available position and then "heapifying" up
- C By sorting the entire heap after each insertion
- D By replacing the largest element if the new element is smaller

Hide Answer

Report

Answer: By placing the new element in the leftmost available position and then "heapifying" up

Explanation

After inserting a new element in a min heap, it's placed in the leftmost available position to maintain the complete tree property. Then, the heap is "heapified" up by comparing the new element with its parents and swapping if necessary until the min heap property is restored, ensuring that parents are always less than or equal to their children.

Q136 A developer finds that their binary heap does not maintain the correct order after several insertions and deletions. What is a likely issue?

- A The heapify process is not correctly implemented
- B The heap is not balanced correctly after operations
- C Keys are not compared correctly during insertions
- D All of the above

Q137 In implementing a trie for a dictionary, a developer notices some words cannot be found. What could be the reason?

- A Nodes for some letters are not correctly linked
- B The search function does not correctly handle word endings
- C Case sensitivity issues
- D A and B

Hide Answer

Report

Answer: Nodes for some letters are not correctly linked

Explanation

If a trie fails to find some words, it may be due to nodes for certain letters not being correctly linked, meaning that the trie structure does not accurately represent the dictionary. It's crucial that each node correctly represents a letter in the word and that nodes are properly linked to reflect the word's spelling.

Q138 What common issue might affect the performance of a splay tree?

- A Frequent splaying of the same nodes
- B Not splaying at every operation
- C Incorrectly balancing the tree
- D Overuse of rotations in splay operations

Hide Answer

Report

Answer: Frequent splaying of the same nodes

Explanation

Frequent splaying of the same nodes can affect the performance of a splay tree. Splay trees rely on moving accessed elements closer to the root through splaying, which can improve average access times. However, if the same nodes are splayed too frequently without accessing a wider range of the tree's elements, it can lead to unbalanced conditions and deteriorate the performance benefits of splaying.

Q143 What is the main idea behind the approximation algorithms?

- A To provide the exact solution to NP-hard problems
- B To provide solutions that are close to the best possible answer for NP-hard problems
- C To reduce the time complexity of algorithms to polynomial time
- D To convert NP-hard problems into P problems

Q144 Why are randomized algorithms used in computing?

- (A) To guarantee the best solution to problems
- (B) To provide a deterministic time complexity for any given problem
- 🟩 (C) To improve the average-case performance of algorithms by introducing randomness
- (D) To simplify the implementation of algorithms

Q145 How do you implement a basic backtracking algorithm for solving the N-Queens puzzle?

- 🟩 (A) By placing queens one by one in different rows and checking for conflicts at each step
- (B) By randomly placing queens on the board and rearranging them to resolve conflicts
- (C) By using a greedy algorithm to place all queens simultaneously
- (D) By calculating the exact positions of all queens before placing them

Q149 Why might a dynamic programming solution perform poorly on a problem with a large state space?

- 🟥 (A) The recursive calls are too deep
- 🟩 (B) The memoization table consumes too much memory
- (C) There are not enough subproblems
- (D) The problem does not exhibit overlapping subproblems

Hide Answer ⓘ Report

Q147 In algorithm design, how is a greedy approach applied to the activity selection problem?

- (A) By selecting activities randomly until no more can be chosen
- (B) By choosing the shortest activities first
- 🟩 (C) By selecting the activities that start the earliest, without overlapping
- (D) By choosing the activities that leave the most free time after completion

Q148 A developer's implementation of a greedy algorithm for a scheduling problem always returns suboptimal solutions. What could be the issue?

- A The algorithm does not consider all possible subsets of tasks
- B The algorithm makes irreversible decisions based on local optima without considering the entire problem
- C The tasks are not sorted correctly before the algorithm is applied
- D The algorithm incorrectly calculates the finish times of tasks

Hide Answer ⓘ Report

Answer: The algorithm makes irreversible decisions based on local optima without considering the entire problem

Explanation

A common issue with greedy algorithms, such as those used for scheduling problems, is that they make decisions based on local optima without considering the entire problem. This can lead to suboptimal solutions if the locally optimal choices do not align with a globally optimal solution. Revising the decision-making criteria or considering a different approach, like dynamic programming, might be necessary.

Q150 In optimizing a recursive algorithm with memoization, a programmer finds that the program runs out of memory. What is a potential solution?

- A Increasing the available memory
- B Converting the recursion to iterative form to use less memory
- C Reducing the problem size
- D Using a more efficient memoization strategy

Hide Answer ⓘ Report