

## 1. What are the applications for BFS, DFS

BFS (Breadth-First Search) and DFS (Depth-First Search) are graph traversal algorithms used to traverse or search through nodes in a graph or tree data structure. Here are some common applications for BFS and DFS:

Applications of BFS:

- Shortest Path and Minimum Spanning Tree: BFS can be used to find the shortest path or the minimum spanning tree between two nodes in an unweighted or weighted graph.
- Crawl the Web: BFS can be used to crawl or scrape web pages from a website in a systematic way. It starts with the root URL and then visits all the linked URLs layer by layer.
- Peer-to-peer Networks: BFS can be used to discover all the reachable nodes in a peer-to-peer network, such as BitTorrent or Gnutella.
- Garbage Collection: BFS can be used to trace all the reachable objects in memory for garbage collection in programming languages like Java.

Applications of DFS:

- Topological Sorting: DFS can be used to perform topological sorting of directed acyclic graphs (DAGs). It helps to order the nodes in a graph based on the dependencies between them.
- Finding Connected Components: DFS can be used to find all the connected components in an undirected graph.
- Solving Mazes: DFS can be used to find a solution to a maze or a puzzle problem. It explores all the possible paths until it reaches the solution.
- Backtracking: DFS can be used for backtracking algorithms like depth-first search in a game tree or a chessboard.

In general, BFS is better suited for finding the shortest path or exploring a graph with uniform weights, while DFS is better suited for topological sorting and exploring deeper paths in a graph.

## 2. Is there any reason why BFS is preferred over DFS in a peer-to-peer network or crawling the web?

BFS and DFS have different strengths and weaknesses, so the choice of algorithm depends on the specific application and the characteristics of the graph or network being traversed.

In the case of crawling the web or exploring a peer-to-peer network, BFS is often preferred over DFS because it ensures that all the reachable nodes are visited in a systematic and predictable order. This is important for applications that require a comprehensive or exhaustive search, such as web crawling or peer-to-peer network discovery.

In contrast, DFS may be preferred over BFS when the goal is to quickly find a solution or a path from a starting node to a target node. DFS is often faster than BFS in finding a solution or a path because it explores deeper paths first, which can lead to earlier detection of a solution or termination of the

search. However, DFS may miss some nodes or paths in the graph if it gets stuck in a deep path or a cycle, which is not a concern for BFS.

Therefore, the choice between BFS and DFS depends on the specific goals and constraints of the application, as well as the characteristics of the graph or network being traversed.

### 3. What are the applications of linked lists?

A linked list is a data structure that consists of a sequence of nodes, each containing a data element and a reference (or pointer) to the next node in the list. Here are some common applications of linked lists:

1. **Implementing Stacks and Queues:** Linked lists can be used to implement common data structures like stacks and queues, where items are added and removed from the same end of the list.
2. **Dynamic Memory Allocation:** Linked lists are often used for dynamic memory allocation because they can grow or shrink in size as needed. This is useful in programming languages like C and C++, where memory must be allocated and deallocated manually.
3. **Implementing Hash Tables:** Linked lists can be used to implement the buckets in a hash table, which is a data structure used for fast lookup and insertion of key-value pairs.
4. **Music and Video Playlists:** Linked lists can be used to implement playlists in music and video players, where each node represents a song or a video and points to the next song or video in the list.
5. **Graphs and Trees:** Linked lists are often used to represent graphs and trees, where each node represents a vertex or a node in the graph and points to its adjacent vertices or child nodes.
6. **Operating Systems:** Linked lists are used extensively in operating systems for various tasks such as maintaining the list of active processes, file system data structures, and managing memory allocation.

Overall, linked lists are a versatile data structure that can be used in a wide range of applications where dynamic storage allocation, sequential access, and efficient insertion and deletion of elements are required.

### 4. What are the applications of stacks?

A stack is an abstract data type that follows the Last-In-First-Out (LIFO) principle, which means that the most recently added item is the first to be removed. Here are some common applications of stacks:

1. **Function Calls and Recursion:** Stacks are used by compilers and programming languages to implement function calls and recursion. When a function is called, the current state of execution is pushed onto the stack, and when the function returns, the previous state is popped from the stack.
2. **Undo and Redo Operations:** Stacks are used to implement undo and redo operations in software applications like text editors and graphic design software. Each time a user performs an action, such as typing a character or moving an object, the state of the document is saved on a stack. If the user wants to undo an action, the previous state is popped from the stack and applied to the document.

3. **Expression Evaluation:** Stacks are used to evaluate arithmetic expressions, such as infix, postfix, and prefix expressions. In postfix notation, also known as Reverse Polish Notation (RPN), operators follow their operands, and each sub-expression is evaluated independently and pushed onto the stack until the final result is obtained.
4. **Browser History:** Stacks are used to implement browser history in web browsers. Each time a user visits a new page, the URL is pushed onto the stack, and when the user clicks the back button, the previous URL is popped from the stack.
5. **Backtracking Algorithms:** Stacks are used to implement backtracking algorithms, such as depth-first search, which explores all possible paths in a search tree and backtracks to the previous state when a dead end is reached.

Overall, stacks are a fundamental data structure that is widely used in computer science and software engineering to implement various algorithms and applications that require efficient last-in-first-out access to data.

## 5. What are the applications of queues?

A queue is an abstract data type that follows the First-In-First-Out (FIFO) principle, which means that the first item added is the first to be removed. Here are some common applications of queues:

1. **Print Spooling:** Queues are used in print spooling systems to store print jobs in the order they were submitted. When a printer is ready to print a job, it dequeues the next job from the queue and starts printing it.
2. **Task Scheduling:** Queues are used in task scheduling systems to store tasks in the order they were submitted or their priority. When a processor is ready to execute a task, it dequeues the next task from the queue based on its scheduling algorithm.
3. **Message Queuing:** Queues are used in message queuing systems to store messages in the order they were received. When a recipient is ready to receive a message, it dequeues the next message from the queue and processes it.
4. **Breadth-First Search:** Queues are used in breadth-first search algorithms to explore all the nodes in a graph or tree level by level. The nodes are added to the queue in the order they are encountered, and dequeued in the same order.
5. **Traffic Management:** Queues are used in traffic management systems to manage the flow of vehicles or people in a queue. When a vehicle or person reaches the front of the queue, they are allowed to proceed, and the next vehicle or person in the queue moves up to the front.

Overall, queues are a fundamental data structure that is widely used in computer science and software engineering to implement various algorithms and applications that require efficient first-in-first-out access to data.

## 6. Can I use BFS for topological sorting?

Yes, you can use Breadth-First Search (BFS) algorithm for topological sorting of a directed acyclic graph (DAG).

The basic idea is to start with the nodes that have no incoming edges (in-degree 0) and push them into a queue. Then, for each node in the queue, visit its adjacent nodes and decrement their in-degree by 1. If a node's in-degree becomes 0, push it into the queue. Repeat the process until all

nodes have been processed. The order in which the nodes are removed from the queue is the topological order of the DAG.

This approach is based on the fact that in a DAG, nodes with no incoming edges are either the source nodes or nodes that have already been processed. Therefore, they can be used as starting points for the topological sorting algorithm.

Note that the topological sort using BFS can have multiple valid orders if there are nodes with the same in-degree. Also, if there is a cycle in the graph, then the algorithm will not be able to generate a valid topological order.

### 7. Can topological sort be used to detect cycles?

Yes, a topological sort can be used to detect cycles in a directed graph. The basic idea is that a directed graph has a cycle if and only if it does not have a valid topological order.

To detect cycles using topological sort, we can run the standard topological sort algorithm, but instead of simply outputting the nodes in order, we keep track of the number of nodes that have been outputted. If at any point during the algorithm, we find that there are no nodes with in-degree 0, but there are still nodes remaining, then the graph has a cycle.

This is because if a node has in-degree 0, then all its incoming edges have already been processed, and it can be safely removed from the graph. Therefore, if there are no nodes with in-degree 0, but there are still nodes remaining, then all the remaining nodes must be part of a cycle.

The time complexity of detecting cycles using topological sort is  $O(V+E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph. This is because we need to visit each vertex and edge at most once during the algorithm.

### 8. What is the maximum length of a path in a directed graph?

The maximum length of a path in a directed graph depends on the specific graph and its structure. In a finite directed graph, the maximum length of a path is bounded by the number of vertices in the graph minus 1, since a path of that length would visit every vertex exactly once.

However, in an infinite directed graph, there may not be a maximum length of a path. For example, in a directed cycle, a path can be arbitrarily long by repeatedly following the edges in the cycle.

It is worth noting that in a directed acyclic graph (DAG), the maximum length of a path can be found using a variant of the topological sorting algorithm. Specifically, we can assign each node a distance value equal to the maximum distance from any source node to that node. We then iterate over the nodes in topological order and update the distance value of each node to be the maximum distance of any of its incoming edges plus 1. The maximum distance value among all nodes in the DAG gives the maximum length of a path in the DAG. The time complexity of this algorithm is  $O(V+E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.

### 9. What is the maximum length of a path in an undirected graph?

The maximum length of a path in an undirected graph also depends on the specific graph and its structure. In a finite undirected graph, the maximum length of a path is bounded by the number of vertices in the graph minus 1, since a path of that length would visit every vertex exactly once.

However, in an infinite undirected graph, there may not be a maximum length of a path. For example, in an infinite cycle, a path can be arbitrarily long by repeatedly following the edges in the cycle.

It is also worth noting that in an undirected graph, there may be multiple paths of maximum length between two vertices. In such cases, any one of these paths can be considered the maximum length path.

In general, finding the maximum length path in an undirected graph is a harder problem than in a directed acyclic graph, since the presence of cycles in the undirected graph makes it difficult to apply the same distance-based algorithm used in a DAG. Depending on the specific graph and the desired output, it may be necessary to use more specialized algorithms or search techniques.

### 10. How to convert a min-heap to a max-heap?

To convert a min-heap to a max-heap, we can simply modify the comparison operation used to compare elements during the heap operations.

In a min-heap, the smallest element is at the root, and during a heap operation, we compare two elements and swap them if the second element is smaller than the first. To convert this to a max-heap, we can simply reverse the comparison so that the largest element is at the root and we swap elements if the second element is larger than the first.

Here are the steps to convert a min-heap to a max-heap:

1. Traverse the min-heap and negate each element (i.e., multiply each element by -1). This will convert the minimum heap property into a maximum heap property, since the largest values will now appear as the smallest negative values.
2. Modify the comparison operation used during heap operations (such as insert, delete, and extract) to compare elements using the standard ">" operator instead of the "<" operator.
3. Traverse the modified max-heap and negate each element again to obtain the original values.

After these steps, the min-heap will be converted into a max-heap. Note that this conversion process does not change the underlying structure of the heap, only the comparison operation used during operations.

### 11. Write python function to convert an array to a max-heap?

Here is an example Python function that converts an array to a max-heap:

The `max_heapify` function takes an array, an index `i` and the size of the heap `n` as input, and modifies the array in-place to maintain the max-heap property at the given index. The `build_max_heap` function

takes an input array, builds a max-heap from it, and returns the resulting heap. It uses `max_heapify` to maintain the max-heap property during the build process.

To use these functions, you can simply call `build_max_heap(arr)` on an input array `arr`. This will return the same array, but with the elements rearranged to form a max-heap.

```
1 def max_heapify(arr, i, n):
2     """
3     Helper function to maintain max-heap property.
4     """
5     l = 2 * i + 1
6     r = 2 * i + 2
7     largest = i
8     if l < n and arr[l] > arr[largest]:
9         largest = l
10    if r < n and arr[r] > arr[largest]:
11        largest = r
12    if largest != i:
13        arr[i], arr[largest] = arr[largest], arr[i]
14        max_heapify(arr, largest, n)
15
16 def build_max_heap(arr):
17     """
18     Builds a max-heap from an input array.
19     """
20     n = len(arr)
21     for i in range(n // 2 - 1, -1, -1):
22         max_heapify(arr, i, n)
23     return arr
24
```

please explain the above step by step

Certainly! Here is a step-by-step explanation of the Python function for converting an array to a max-heap:

The `max_heapify` function takes three arguments:

- `arr`: the input array to be converted to a max-heap
- `i`: the index of the current element to be checked/maintained
- `n`: the size of the heap

The function starts by calculating the indices of the left and right children of the current element using the formulas  $2 * i + 1$  and  $2 * i + 2$ , respectively.

Next, the function initializes a variable `largest` to the index of the current element `i`. It then checks whether either of the child nodes is larger than the current node, and if so, updates `largest` accordingly.

If the largest element is not the current element, the function swaps the two elements and then recursively calls itself on the largest child node to maintain the max-heap property there.

The `build_max_heap` function uses the `max_heapify` function to build a max-heap from an input array:

This function takes an input array `arr`, initializes the size of the heap `n` to the length of the array, and then iterates over the array in reverse order starting from the middle. For each element in the array, it calls `max_heapify` to maintain the max-heap property at that index.

By iterating in reverse order from the middle, we ensure that every element is checked and potentially swapped with its children, which guarantees that the final result is a valid max-heap.

Finally, the function returns the resulting max-heap.

To use these functions, you can simply call `build_max_heap(arr)` on an input array `arr`. This will return the same array, but with the elements rearranged to form a max-heap.

## 12. I can't understand why the looping starts from middle of the array?

Starting the loop from the middle of the array is an optimization technique that can make the build heap process more efficient. Here's why:

The heap structure can be visualized as a binary tree where each node has at most two child nodes. In a complete binary tree, all levels are completely filled except possibly the last level, which is filled from left to right.

When we convert an array to a heap, we are essentially building a binary tree where each node is an element of the array. To do this efficiently, we want to minimize the number of times we call `max_heapify` on each node in the tree.

Consider the last level of the tree, which contains only the leaves. Since each node in this level has no children, there is no need to call `max_heapify` on any of these nodes. Therefore, we can start our loop from the last non-leaf node, which is located at index  $n // 2 - 1$ , where `n` is the length of the array.

By starting our loop from the middle of the array, we avoid unnecessarily calling `max_heapify` on nodes that are already leaves, which can improve the performance of the algorithm.

Note that this optimization is not strictly necessary, and it may not make a significant difference in practice for small arrays. However, for large arrays, this optimization can make a noticeable difference in the running time of the algorithm.

## 13. 4 sorted lists each of length $n/2$ are merged into a single sorted list of $2n$ elements using two way merging. What will be the minimum number of element comparisons needed for this process ?

When merging two sorted lists of lengths `m` and `n`, the minimum number of element comparisons needed is  $m + n - 1$ .

Using this fact, we can compute the minimum number of comparisons needed to merge four sorted lists each of length  $n/2$  into a single sorted list of  $2n$  elements using two-way merging as follows:

- First, we merge the first two lists of length  $n/2$  to create a sorted list of length `n`. The minimum number of comparisons needed for this step is  $(n/2) + (n/2) - 1 = n - 1$ .

- Next, we merge the third and fourth lists of length  $n/2$  to create another sorted list of length  $n$ . The minimum number of comparisons needed for this step is also  $n - 1$ .
- Finally, we merge the two sorted lists of length  $n$  using two-way merging. The minimum number of comparisons needed for this step is  $2n - 1$ .

Therefore, the minimum number of comparisons needed to merge the four sorted lists into a single sorted list of  $2n$  elements using two-way merging is:

$$(n - 1) + (n - 1) + (2n - 1) = 4n - 3$$

So the answer is  $4n - 3$ .

#### 14. Please explain the following code

```
1 class maxheap:
2     def __init__(self):
3         self.A = []
4     def max_heapify(self,k):
5         l = 2 * k + 1
6         r = 2 * k + 2
7         largest = k
8         if l < len(self.A) and self.A[l] > self.A[largest]:
9             largest = l
10        if r < len(self.A) and self.A[r] > self.A[largest]:
11            largest = r
12        if largest != k:
13            self.A[k], self.A[largest] = self.A[largest], self.A[k]
14            self.max_heapify(largest)
```

The code above defines a Python class `maxheap` for implementing a max-heap. A max-heap is a binary tree where the value of each node is greater than or equal to the values of its children. The `max_heapify` method is used to maintain the max-heap property after inserting or removing an element from the heap.

Let's break down the code step by step:

```
class maxheap:
    def __init__(self):
        self.A = []
```

This defines a class `maxheap` with an attribute `A` that will be used to store the heap elements. The `__init__` method initializes the heap to an empty list.

```
def max_heapify(self,k):
    l = 2 * k + 1
    r = 2 * k + 2
    largest = k
```

This method takes an index `k` as input and assumes that the binary trees rooted at `left(k)` and `right(k)` are max-heaps, but that the binary tree rooted at `k` may not be a max-heap. The method



then compares the value at index `k` with the values at indices `left(k)` and `right(k)` to find the largest value among the three. The variable `largest` stores the index of the largest value.

```
if l < len(self.A) and self.A[l] > self.A[largest]:
    largest = l
if r < len(self.A) and self.A[r] > self.A[largest]:
    largest = r
```

These `if` statements check whether the left and right children of node `k` exist and have larger values than the value at index `largest`. If so, `largest` is updated to the index of the larger child.

```
if largest != k:
    self.A[k], self.A[largest] = self.A[largest], self.A[k]
    self.max_heapify(largest)
```

If the value at index `k` is not the largest value among `k`, `left(k)`, and `right(k)`, then the values at indices `k` and `largest` are swapped. This ensures that the value at index `k` is greater than or equal to its children. The method then recursively calls itself with `largest` as the input index, to ensure that the subtree rooted at index `largest` also satisfies the max-heap property.

Overall, the `max_heapify` method takes  $O(\log n)$  time to run, where `n` is the number of elements in the heap, because it moves down the heap by one level at each iteration of the while loop.

### 15. In a minimal spanning tree, how to find missing edge lengths

In a minimum spanning tree, all the edges that are selected to be a part of the tree have a length assigned to them. The length of the edge represents the weight of the edge in the graph.

If there are edges missing in the minimum spanning tree, it means that they were not selected to be a part of the tree, and hence their weight is not known. To find the weight of these missing edges, you would need to consider all the edges that are not a part of the minimum spanning tree and calculate their weights.

To do this, you could use a graph traversal algorithm such as depth-first search (DFS) or breadth-first search (BFS) to explore all the edges in the graph that are not a part of the minimum spanning tree. As you traverse the graph, you can calculate the weights of the missing edges by summing up the weights of the edges that are traversed.

Alternatively, you could use an algorithm like Kruskal's algorithm or Prim's algorithm to compute the minimum spanning tree, and then calculate the weights of the missing edges as the difference between the weight of the original graph and the weight of the minimum spanning tree.

### 16. Can you give an example

Sure, here is an example:

Suppose you have a graph with the following edges and weights:

```
(1, 2) = 3
(1, 3) = 2
(2, 3) = 4
(2, 4) = 1
(3, 4) = 5
(3, 5) = 6
(4, 5) = 7
```

To find the minimum spanning tree of this graph, you can use Kruskal's algorithm or Prim's algorithm. For simplicity, let's use Prim's algorithm:

1. Choose an arbitrary vertex, say vertex 1, to be the root of the tree.
2. Add all the edges that connect vertex 1 to its neighbors (vertices 2 and 3) to a priority queue.
3. Pop the edge with the smallest weight from the priority queue and add it to the tree. In this case, it is the edge between vertex 1 and vertex 3 with weight 2.
4. Add all the edges that connect vertex 3 to its neighbors (vertices 1, 2, 4, and 5) to the priority queue, but only if they connect to vertices that are not already in the tree.
5. Pop the edge with the smallest weight from the priority queue and add it to the tree. In this case, it is the edge between vertex 3 and vertex 2 with weight 4.
6. Add all the edges that connect vertex 2 to its neighbors (vertices 1, 3, and 4) to the priority queue, but only if they connect to vertices that are not already in the tree.
7. Pop the edge with the smallest weight from the priority queue and add it to the tree. In this case, it is the edge between vertex 2 and vertex 4 with weight 1.
8. Add all the edges that connect vertex 4 to its neighbors (vertices 2, 3, and 5) to the priority queue, but only if they connect to vertices that are not already in the tree.
9. Pop the edge with the smallest weight from the priority queue and add it to the tree. In this case, it is the edge between vertex 4 and vertex 5 with weight 7.

The resulting minimum spanning tree has the following edges and weights:

```
(1, 3) = 2
(3, 2) = 4
(2, 4) = 1
(4, 5) = 7
```

To find the missing edge lengths, we can subtract the weight of each edge in the minimum spanning tree from the total weight of the original graph:

```
Total weight of original graph = 3 + 2 + 4 + 1 + 5 + 6 + 7 = 28
Total weight of minimum spanning tree = 2 + 4 + 1 + 7 = 14
Missing edge lengths:
(1, 2) = 3 - 0 = 3
(2, 3) = 4 - 0 = 4
(3, 5) = 6 - 0 = 6
```

### 17. What are the rules for adjacency matrix of graph in transitive closure?

In a directed graph, the transitive closure of a graph represents the direct and indirect paths between every pair of vertices.

The rules for adjacency matrix of a graph in transitive closure are as follows:

1. The diagonal elements of the matrix are set to 1 since there is always a path from a vertex to itself.
2. For any pair of vertices  $(i, j)$ , if there is a direct edge from  $i$  to  $j$ , then the  $(i, j)$  element of the matrix is set to 1.
3. For any pair of vertices  $(i, j)$ , if there is no direct edge from  $i$  to  $j$ , then the  $(i, j)$  element of the matrix is set to 1 if there exists a path from  $i$  to  $j$  through one or more vertices in the graph. Otherwise, the  $(i, j)$  element of the matrix is set to 0. This can be calculated by taking the logical OR of the  $(i, k)$  and  $(k, j)$  elements of the matrix for all vertices  $k$  in the graph.

By applying these rules, we can construct the adjacency matrix of a graph in transitive closure.

### 18. How can we use the Floyd-Warshall algorithm for all-pairs shortest paths to detect whether a graph has a negative cycle?

The Floyd-Warshall algorithm is a dynamic programming algorithm that computes the shortest path between all pairs of vertices in a directed, weighted graph. It can also be used to detect the presence of negative cycles in the graph.

To detect whether a graph has a negative cycle using the Floyd-Warshall algorithm, we follow these steps:

1. Initialize the distance matrix  $D$  to hold the weights of the edges between each pair of vertices in the graph.
2. For each vertex  $i$ , and for each other vertex pair  $(j, k)$ , update the distance matrix  $D$  as follows:  
$$D[j][k] = \min(D[j][k], D[j][i] + D[i][k])$$

This means that if there is a shorter path from vertex  $j$  to vertex  $k$  that passes through vertex  $i$ , then we update the distance between vertices  $j$  and  $k$  to be the length of that shorter path.
3. After completing step 2, check the diagonal elements of the distance matrix. If any diagonal element  $D[i][i]$  is negative, then there exists a negative cycle in the graph.

The reason this works is that the Floyd-Warshall algorithm computes the shortest path between all pairs of vertices by iteratively updating the distance matrix. If there is a negative cycle in the graph, then the algorithm will eventually detect it because the distances will continue to decrease indefinitely as the algorithm iterates.

Note that if there is no negative cycle in the graph, then the algorithm will terminate with the correct shortest path distances between all pairs of vertices.

19. Does prim's algorithm and kruskal's algorithm produce same trees always? Under what condition, will they produce different trees?

Under normal circumstances, when the edge weights are unique, both algorithms will produce the same minimum spanning tree. However, there are some cases where they may produce different trees:

1. When the graph has multiple edges with the same weight: In this case, the choice of which edge to add first can be different in Prim's algorithm and Kruskal's algorithm, leading to different minimum spanning trees.
2. When the graph is not connected: Both algorithms require the graph to be connected in order to compute a minimum spanning tree. If the graph is not connected, then the algorithms will produce a minimum spanning forest, which may not be unique.

20. Is a spanning tree a connected acyclic graph?

Yes, a spanning tree is a connected acyclic graph.

A spanning tree of an undirected graph is a subgraph that includes all the vertices of the original graph and is also a tree, which means it is a connected acyclic graph. In other words, a spanning tree is a tree that spans (i.e., covers) all the vertices of the original graph.

Similarly, a spanning tree of a directed graph is a subgraph that includes all the vertices of the original graph and is also a directed acyclic graph (DAG), which means it is a connected acyclic graph with directed edges. In other words, a spanning tree of a directed graph is a DAG that spans all the vertices of the original graph.

21. Will adding an edge to a spanning tree always result in a cycle in the graph?

Yes, adding an edge to a spanning tree will always result in a cycle in the graph.

If we add an edge to a spanning tree, we are adding a new edge to the graph that connects two vertices that were not previously connected in the tree. This new edge creates a cycle in the graph because it forms a path that starts and ends at the same vertex, which means it has to traverse at least one edge twice.

22. In the Floyd-Warshall algorithm output, what are the possible values of diagonal elements?

The diagonal represents the shortest path from a node to itself, which is always zero. If there are non-zero values on the diagonal in the output matrix, then it indicates that there is a negative cycle in the graph.

23. What is the total number of spanning trees possible from a graph?

The total number of spanning trees possible from a graph can be calculated using Kirchhoff's theorem, which states that the number of spanning trees of a graph is equal to any cofactor of the Laplacian matrix of the graph. The Laplacian matrix of a graph is defined as follows:

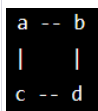
- The diagonal elements of the Laplacian matrix are the degrees of the vertices (i.e., the number of edges incident on the vertex).

- The off-diagonal elements of the Laplacian matrix are either 0 or -1, depending on whether the vertices are adjacent or not.

Using the Laplacian matrix, the number of spanning trees can be calculated as follows:

- Choose any row or column of the Laplacian matrix.
- Delete the corresponding row and column from the Laplacian matrix to obtain a smaller matrix.
- Calculate the determinant of the smaller matrix.
- The number of spanning trees is equal to the absolute value of the determinant.

For example, consider the following graph:



The Laplacian matrix of the graph is:

$$\begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix}$$

If we choose the first row, and delete the first row and column, we get the following smaller matrix:

$$\begin{bmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

The determinant of this matrix is 6, so the total number of spanning trees is 6.

#### 24. What is the difference between heapify and building a heap?

Heapify is a process of maintaining the heap property of a heap. It performs a comparison of the root with its left and right children and swaps the node with the largest value to the root position, recursively heapifying the affected subtree(s).

On the other hand, building a heap involves creating a heap from an unsorted array. It does this by repeatedly calling heapify on each node, starting from the bottom-most right-most node and working its way up to the root node.

#### 25. What is the maximum number of edges in DAG with n vertices?

$n(n-1)/2$ .

26. What is the maximum number of edges in DAG with  $n$  vertices?

$n - 1$

27. What is the difference between a complete graph and a DAG, both seem to have same maximum number of edges.

The main difference between a complete graph and a DAG is that a complete graph is an undirected graph in which every pair of vertices is connected by an edge, while a DAG is a directed graph that has no directed cycles. Although they have the same maximum number of edges, they are used to model different types of relationships and have different properties.

28. What is the difference between a complete graph and a connected graph.

Complete graph is a graph in which all vertices are directly connected to each other, while a connected graph is a graph in which all vertices can be reached from any other vertex by traversing a sequence of edges.

29. What is the order of a graph with  $n$  vertices and  $m$  edges?

Order of a graph is simply the number of vertices in the graph.

30. What is the size of a graph with  $n$  vertices and  $m$  edges?

Size of a graph is simply the number of edges in the graph.

31. Bellman-Ford is run for  $(n - 1)$  iterations, each time identifying paths that could reach each vertex through one additional hop. Is this statement true or false?

Yes, during each iteration, the algorithm relaxes all the edges in the graph, which updates the shortest path estimates of each vertex by considering paths with one additional hop.

However, for some vertices algorithm may converge earlier than  $(n-1)$  iterations, depending on the structure of the input graph. Further, if a negative weight cycle is present, then the algorithm reports that there is no shortest path since the total weight of the path can be made arbitrarily small by traversing the cycle repeatedly.

32. Consider a list  $L$  with  $k$  distinct numbers and a heap  $H$  with size  $n$ . What is the nearest upper bound for checking if every number in  $L$  is present in the heap  $H$  ?

For searching any element in heap, you need to traverse all elements of the heap or in some cases  $O(N/2)$  but ultimately we come to  $O(N)$  and for  $k$  distinct element it will be  $O(kN)$ .

33. Consider a max-heap  $H$  with  $n$  elements and height  $h$ . What is the nearest upper bound to remove the maximum element from max-heap  $H$ ?

$O(\log n)$  or  $O(h)$

34. Suppose we implement merge sort with a five-way split: divide the array into 5 equal parts, sort each part and do a 5 way merge. What would the worst-case complexity of this version be?

$O(n \log n)$

35. Divide the problem into 6 sub-problems, each of size  $6n$ . Number of steps required to combine these 6 solutions is  $2n + 12$ . We apply this strategy recursively till the sub-problems can not be further divided into sub-problems.

$O(n \log n)$

36. If we divide the problem into 4 sub-problems of size  $2n$  and number of steps required to combine the solutions is 15 using some optimizations, what will be the nearest upper bound of this algorithm?

$O(n^2)$

Discussions on questions from past papers

<https://discourse.onlinedegree.iitm.ac.in/t/quiz-1-revision-question/67780/2>

<https://discourse.onlinedegree.iitm.ac.in/t/quiz-1-question-88-query/68201>

<https://discourse.onlinedegree.iitm.ac.in/t/q92-quiz-1/68048>

<https://discourse.onlinedegree.iitm.ac.in/t/q-96-quiz-1/68049>

<https://discourse.onlinedegree.iitm.ac.in/t/previous-year-qp/67591>

<https://discourse.onlinedegree.iitm.ac.in/t/previous-quiz-1-q11/67301/3>

<https://discourse.onlinedegree.iitm.ac.in/t/previous-quiz-q-99/67005>

<https://discourse.onlinedegree.iitm.ac.in/t/binary-search-previous-term-question/67096>

<https://discourse.onlinedegree.iitm.ac.in/t/avl-tree/33842/5>

## Problems

1. Consider a directed graph  $G$  with 90 edges with the least number of vertices possible. What will be the number of vertices in graph  $G$ .

For a directed graph  $G$  with 90 edges and the least number of vertices possible, the number of vertices can be found by considering the minimum number of vertices required to support 90 directed edges.

Note that in a directed graph, each edge goes from one vertex (the tail of the edge) to another vertex (the head of the edge). Therefore, if there are  $n$  vertices in the graph, the maximum number of directed edges that can be present is  $n(n-1)$  (assuming that there are no self-loops).

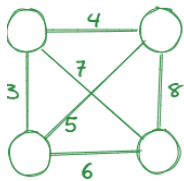
To have a graph with 90 edges and the least number of vertices possible, we want to find the smallest integer value of  $n$  such that  $n(n-1)$  is greater than or equal to 180 (since each directed edge contributes to the total count twice - once for the head and once for the tail).

Thus,  $n(n-1) \geq 180$

Solving the inequality, we get  $n = 14$  is the minimum number of vertices.

2. Let  $G$  be a complete undirected graph on 4 vertices, having 6 edges with weights being 3, 4, 5, 6, 7 and 8. What is the maximum possible total weight that a minimum weight spanning tree of  $G$  can have?

In this question, note that there could be multiple graphs with vertices having 6 edges. MST created out of one of these graphs would yield maximum total weight. Look at the following graph.



In this case, once the lowest two weights are selected into the MST, the third weight 5 is not selected, since it forms a cycle. So, it must select the next lowest, which is 6.

Thus, the total weight is  $3 + 4 + 6 = 13$

3. Following nodes are inserted into an AVL tree in the same order - 36,40,32,18,72,5,35,34. Which ones are the leaf nodes?

This is a seemingly easy problem, but there's a tendency for overlooking node '32' needing rebalancing in the penultimate step. This makes 32 as one of the leaf nodes. The other leaf nodes are 5, 35 and 72.