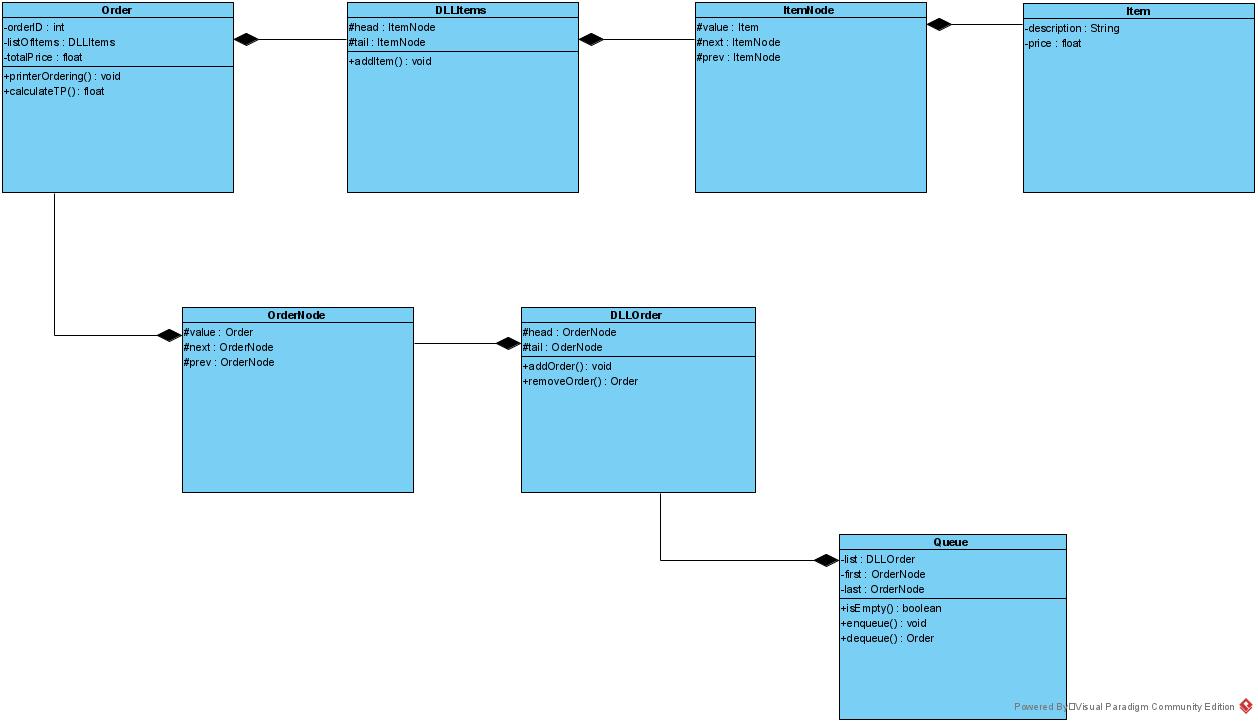
**Student Assessment Submission and Declaration**

When submitting evidence for assessment, each student must sign a declaration confirming that the work is their own.

|  |  |  |  |
| --- | --- | --- | --- |
| Student name: Rashed Hasan Qahah | | Assessor name: | |
| Issue date (1St Submission): | Submission date (1St Submission): | | Submitted on: |
| In case of resubmission | | | |
| Issue date (1St Submission): | Submission date (1St Submission): | | Submitted on: |
| Programme: | | | |
|  | | | |
| Assignment number and title: | | | |

Task 1: Handling Orders

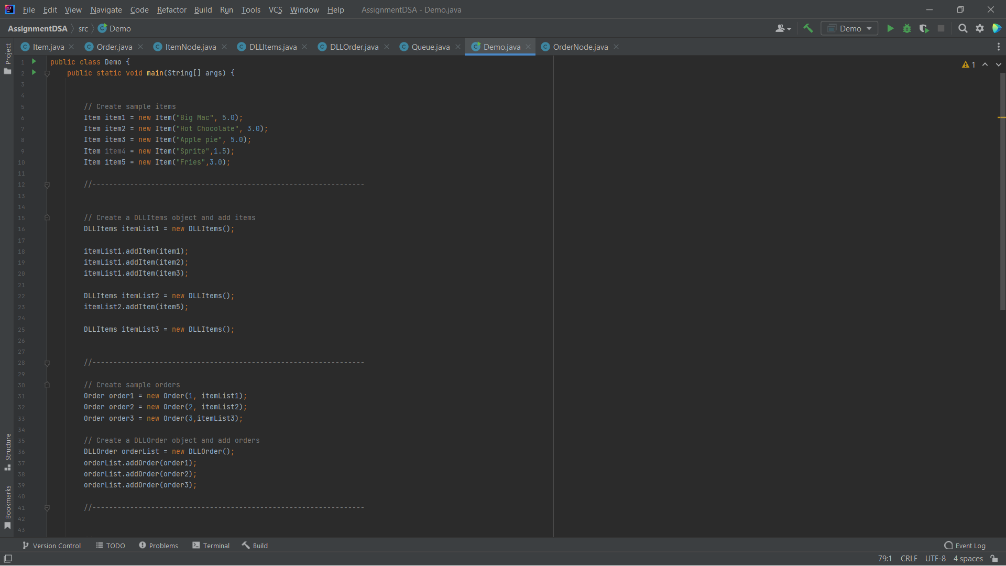
1.1: Draw a class diagram that includes the following classes and show the relationship between them, information hiding, and ADT specifications:



**1.2:** Implement the solution according to the given classes and specifications, and prevent handling orders when there are no orders in queue and prevent creating invoice/total price for empty list of items.

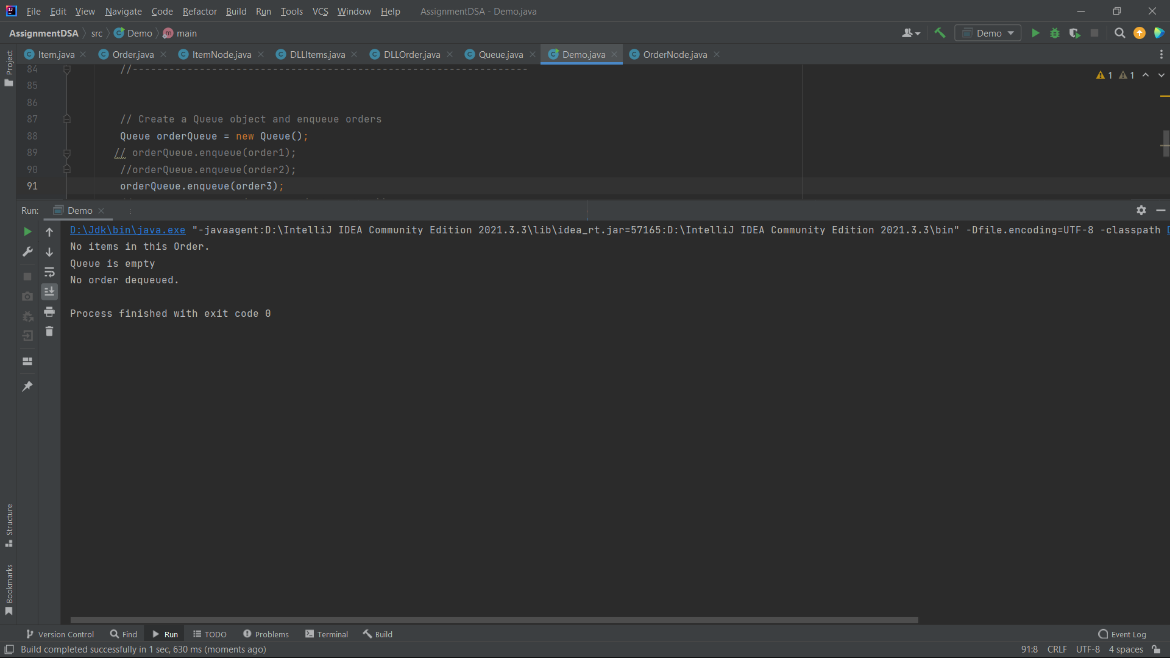
**Answer:**

**- Class Demo:**

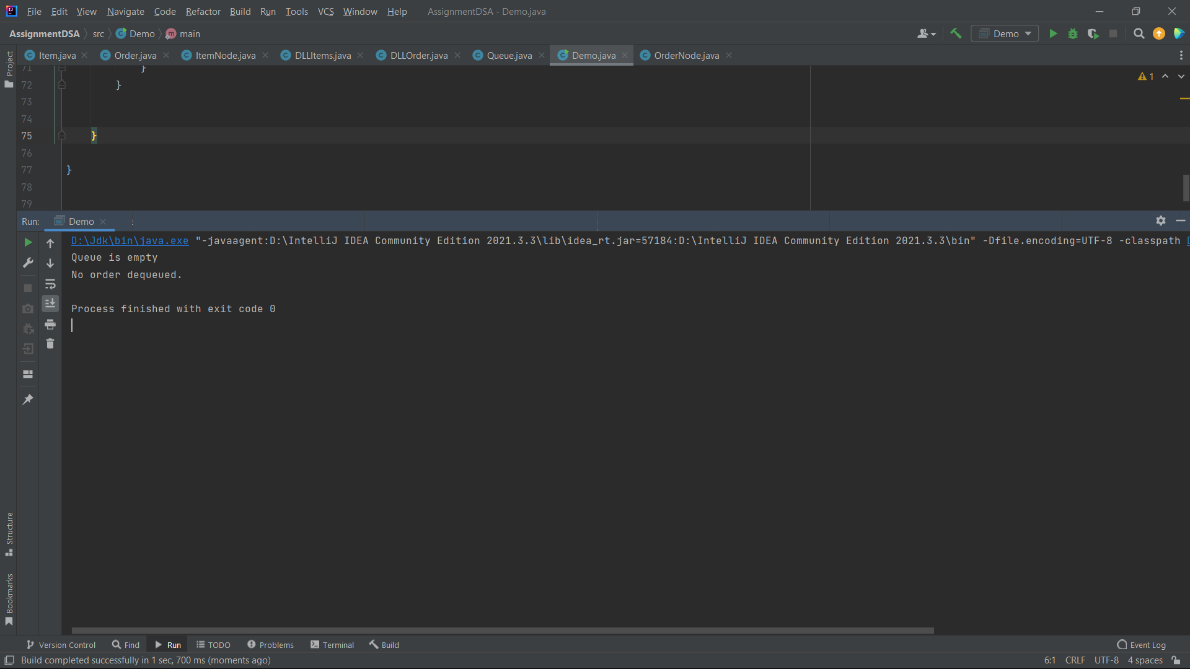


**- Run Output:**

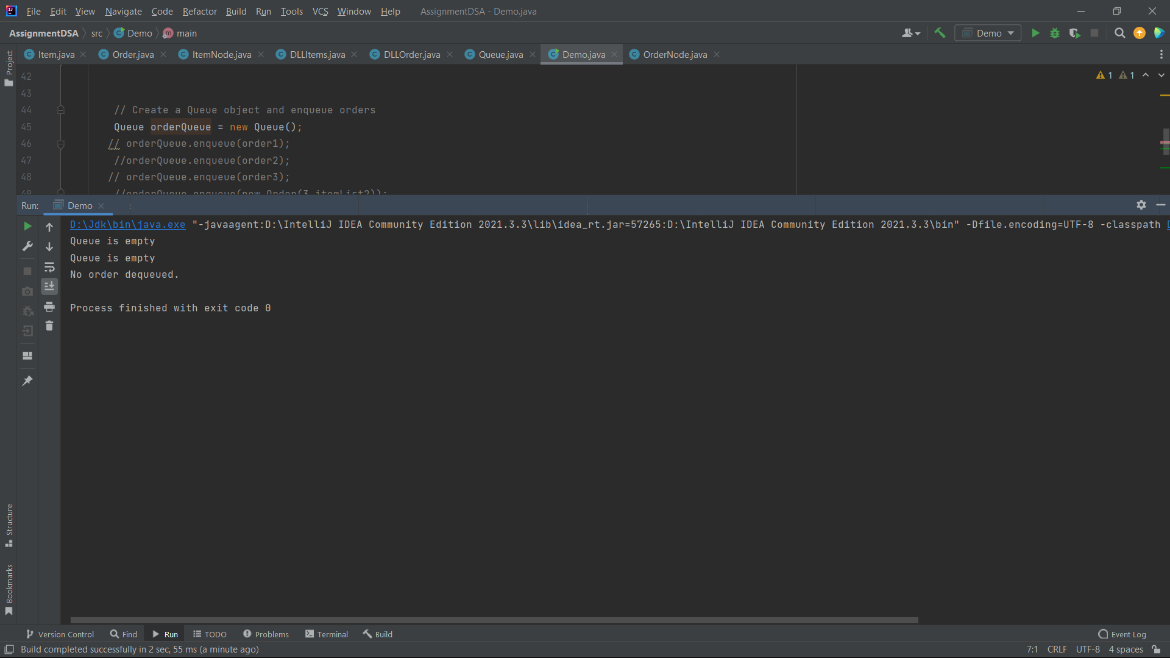
**- 1) orderQueue.enqueue(order3);**

****

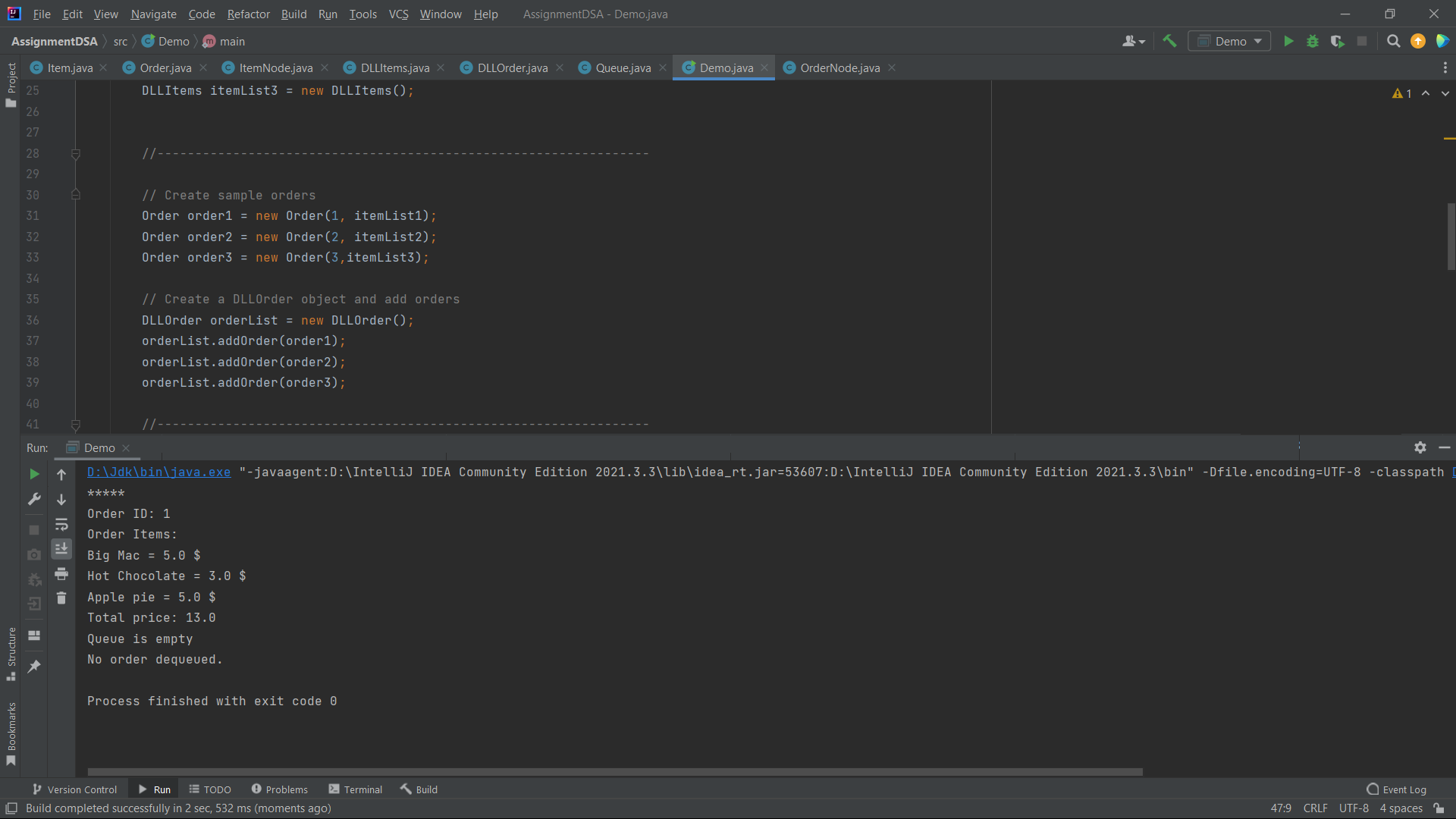
**-2) We did not add any Order in Queue.**

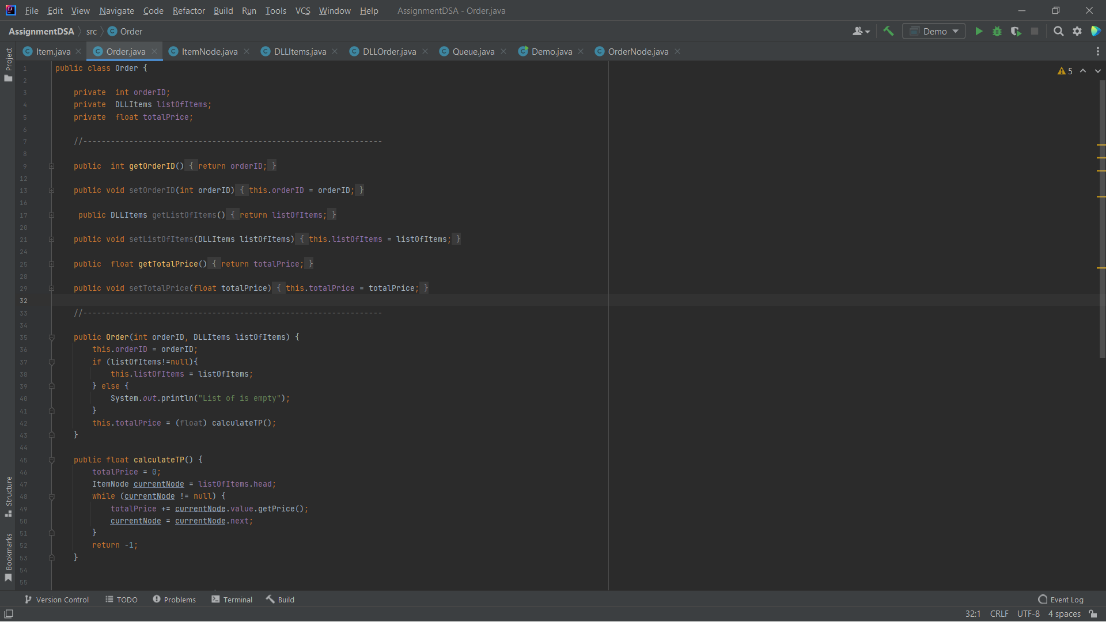
****

**-3) orderQueue.dequeue();**

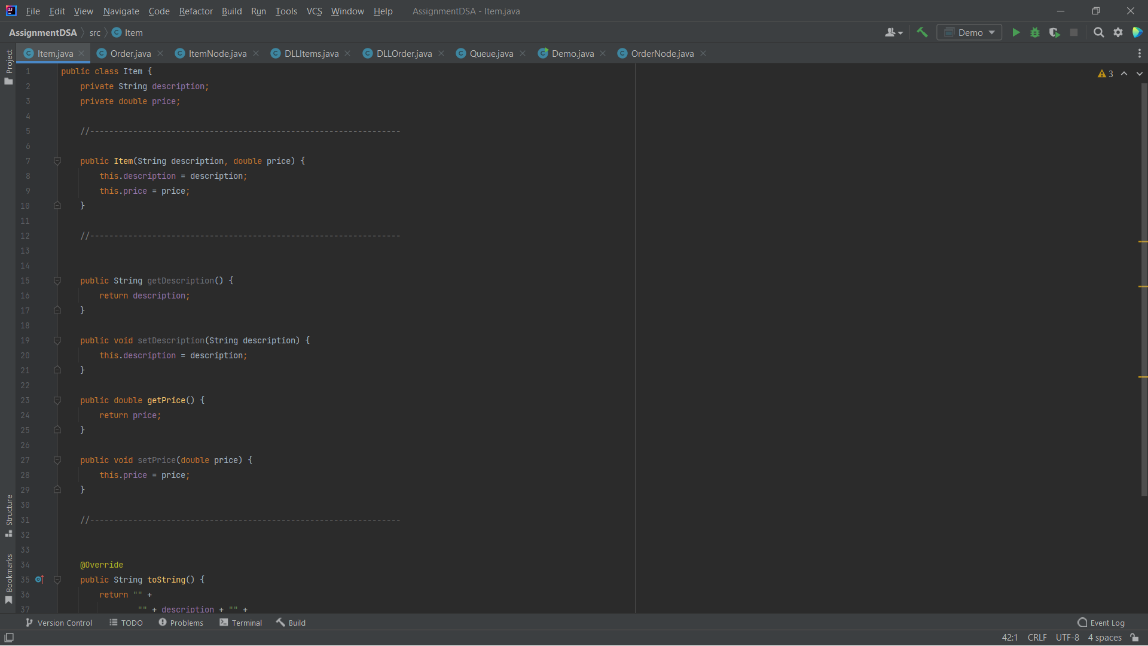
****

**-4) orderQueue.enqueue(order1);**

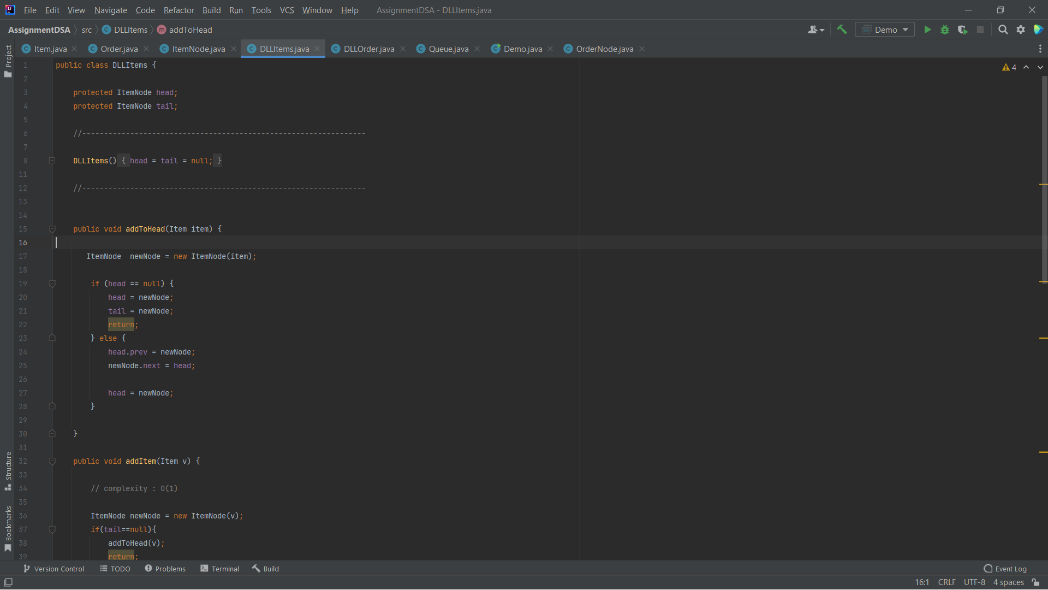


**- 1) Class Order:**   
****

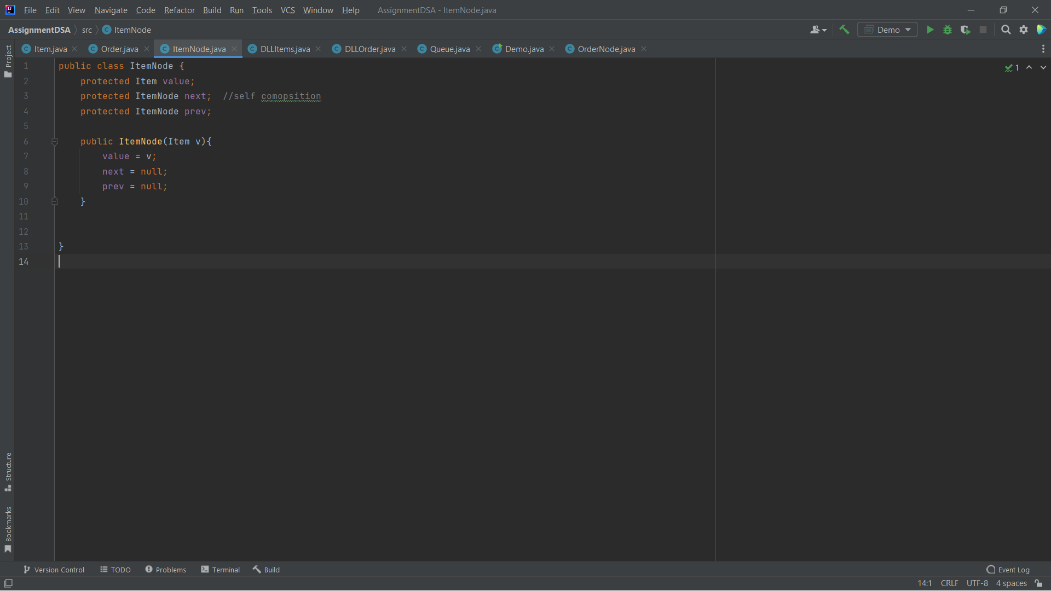
**-2) Class Item:**

****

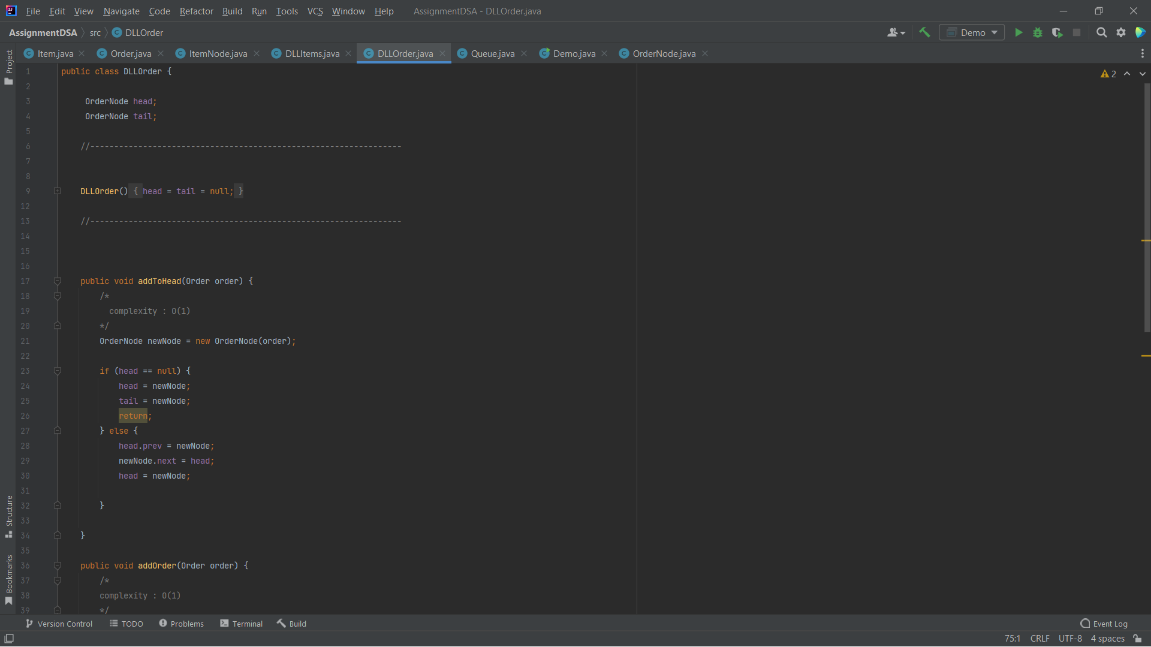
**-3) Class DLLItem:**

****

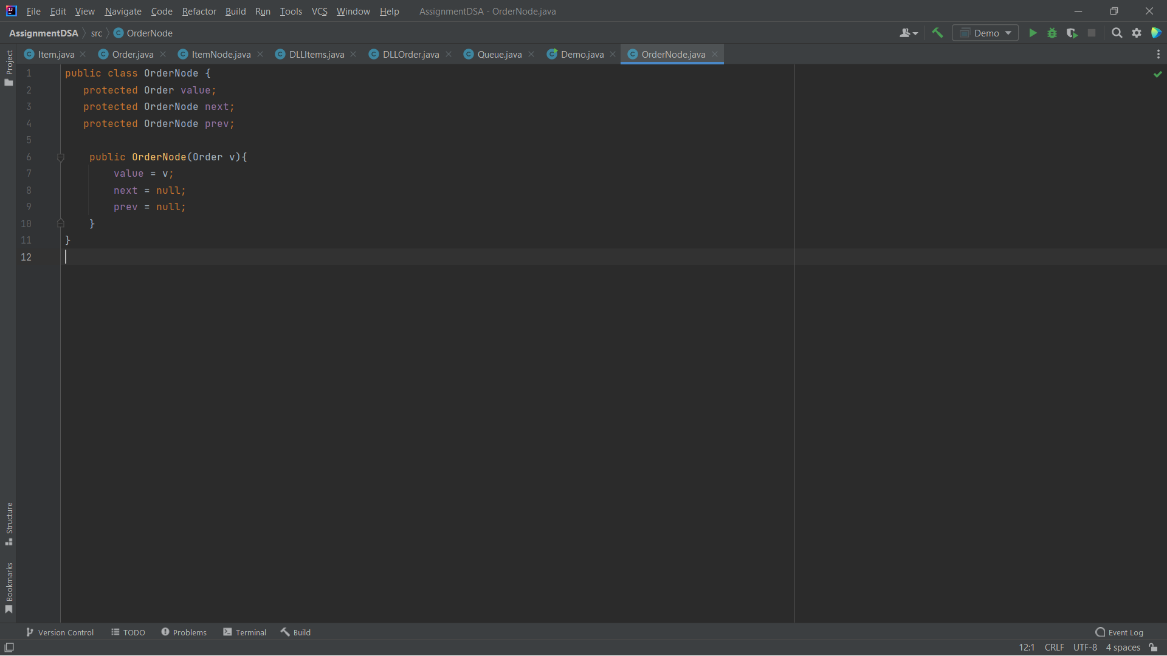
**-4) Class ItemNode:**

****

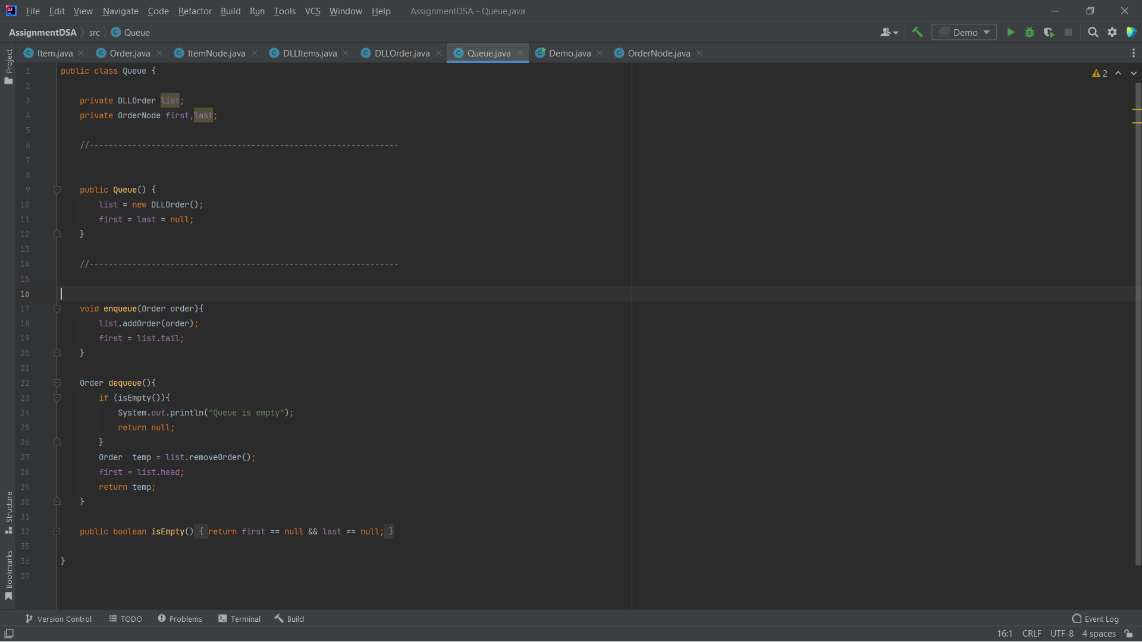
**-5) Class DLLOrder:**

****

**-6) Class OrderNode:**

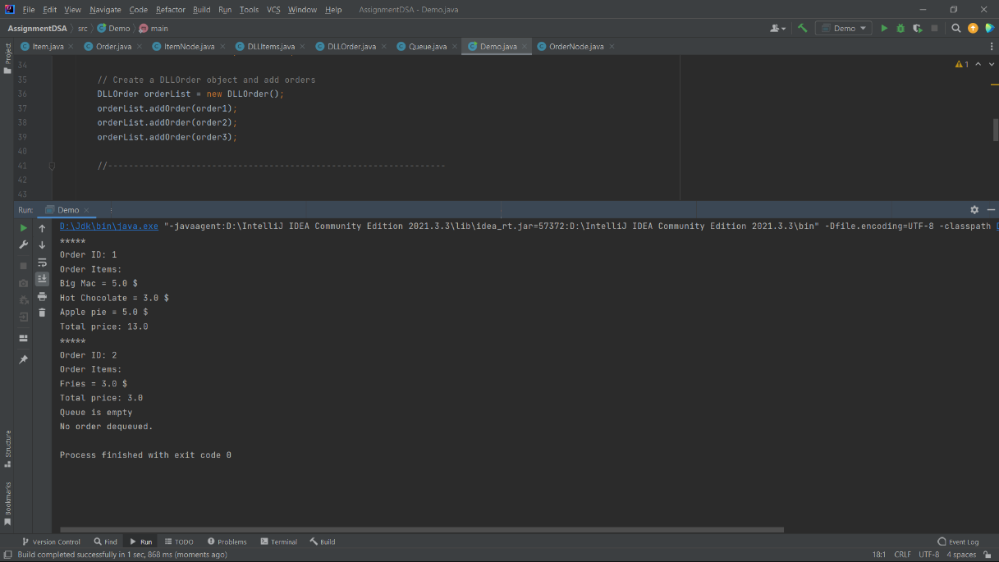
****

**-7) Class Queue**

****

**1.3:** Test your implementation using a Demo class, Figure 1 shows a sample testing. **Present** how the specified ADT solves the given problem.

**Answer:**

****

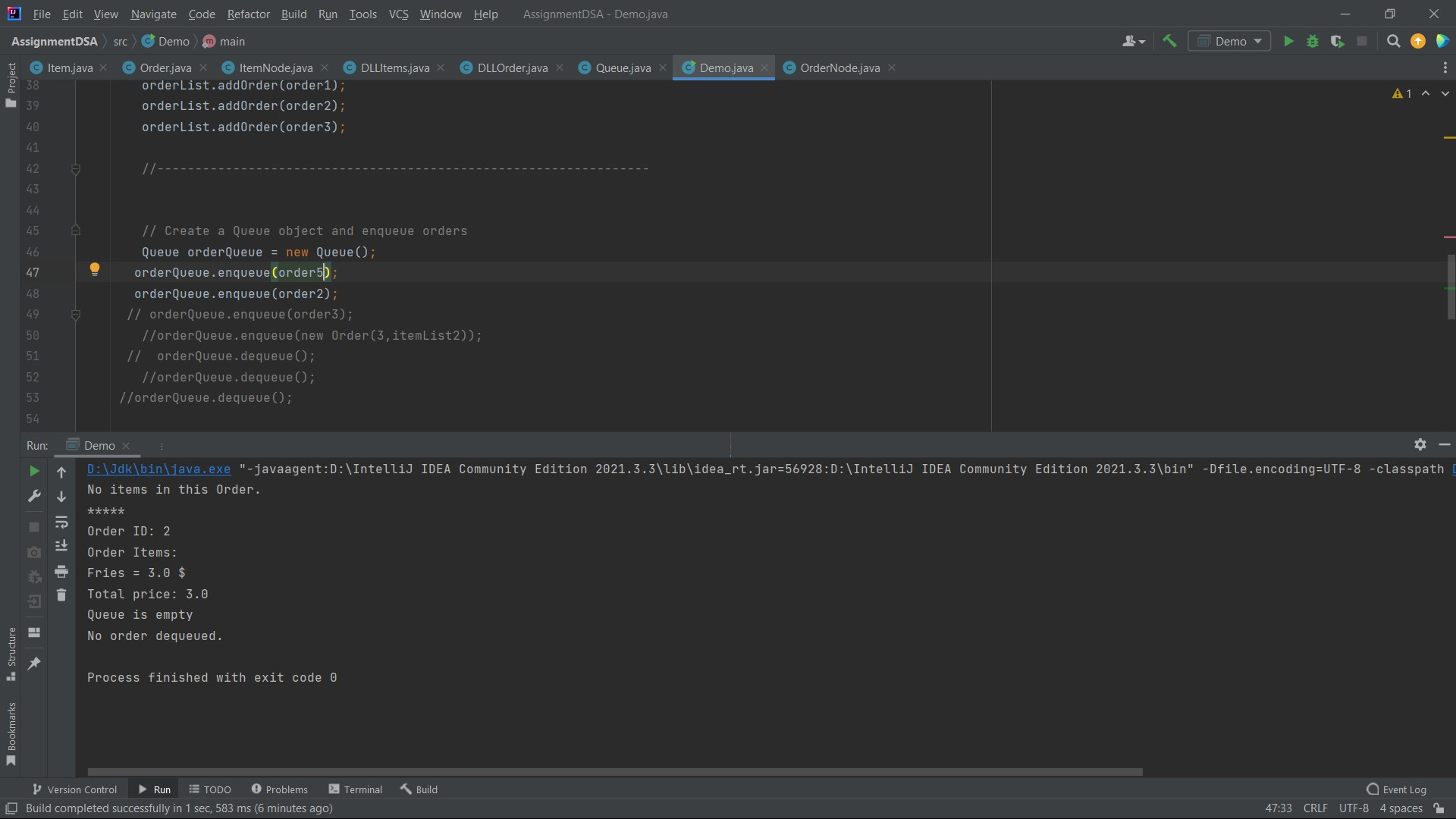
Here, Order 1 Details, the first printed block in the output corresponds to the details of the first order that was enqueued and then dequeued. It contains the order ID (1), the list of items in the order (Big Mac, Hot Chocolate, and Apple pie), and the total price of the order (13.0). The prices of individual items are also printed. This happens in the printerOrdering () method of the Order class.

Also, Order 2 Details: The next block in the output corresponds to the second order that was enqueued and then dequeued. Similar to the first one, it displays the order ID (2), the item in the order (Fries), and the total price (3.0).

Empty Queue: After the second order has been dequeued, an attempt is made to dequeue another order from the queue. However, because the queue is empty at this point, the output "Queue is empty" is printed. This message is produced by the dequeue () method in the Queue class when there are no more orders in the queue.

No Order Dequeued: Finally, because the queue was empty and no order could be dequeued, the message "No order dequeued" is printed, as seen in the while loop of the main method in the Demo class. It's trying to print the details of an order that doesn't exist, hence the null check results in this output.

This demonstrates the use of a queue to handle orders, and how it can be used to manage and process them in a First-In-First-Out (FIFO) manner. The enqueue operation is used to add orders to the queue, while dequeue is used to remove orders for processing. Once an order is processed (i.e., dequeued and its details printed), it's removed from the queue. This process repeats until the queue is empty, at which point the application indicates that there are no more orders to process.



Here, "No items in this Order.": This message appeared because the first order that was dequeued from the queue (order5) had no items in it. order5 was created with an empty DLLItems object which means it doesn't contain any Item instances. When this order is dequeued and its details are printed, the program checks the DLLItems object within the Order, finds it empty, and thus prints out "No items in this Order."

Order ID: 2, Order Items: Fries = 3.0 $, Total price: 3.0: This part of the output is the result of dequeuing the second Order in the queue (order2). order2 had one item (Fries), so it printed the order's ID, the item and its price, and the total price of the order.

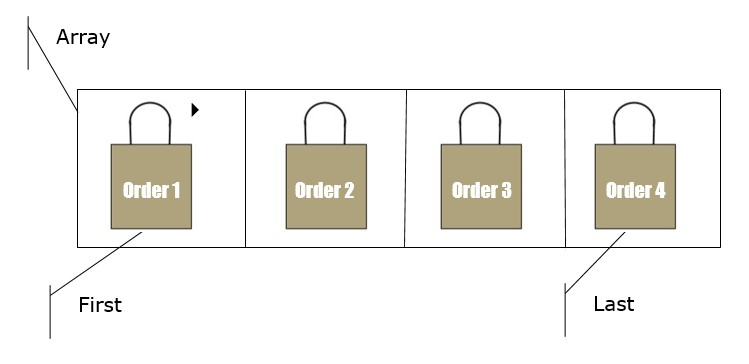
"Queue is empty": After the program dequeued order2, there were no more Order objects left in the Queue. So, when it attempted to dequeue another Order, it detected that the queue was empty and printed out this message.

"No order dequeued.": The program tried to dequeue another Order object after order2, but since the queue was already empty, there was no order to dequeue. This resulted in the message "No order dequeued."

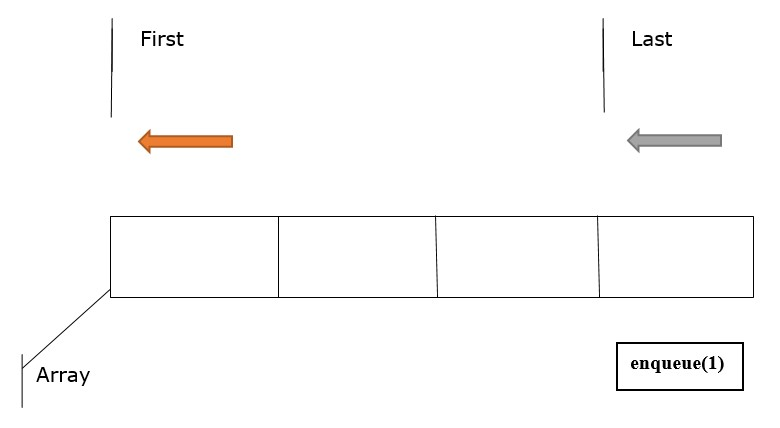
**1.4:** **Draw** an illustration to show how queue handles the orders in FIFO manner, give examples.

**Answer:**

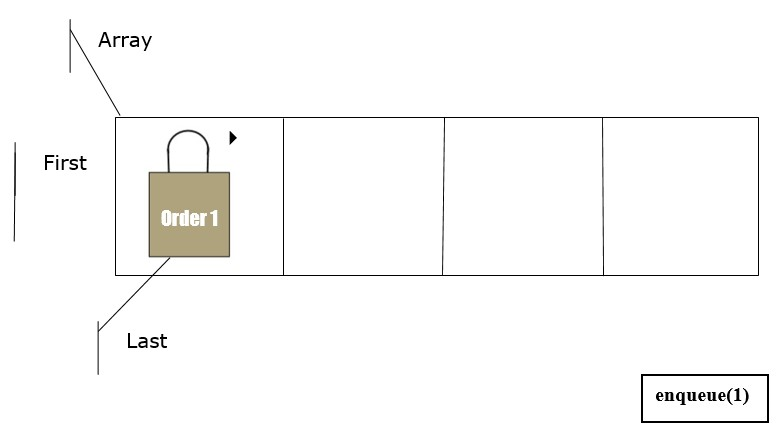
The restaurant has 4 orders, and they can order from the restaurant with drive-thru, so the first order that enters the drive-thru service is the first order that leaves.



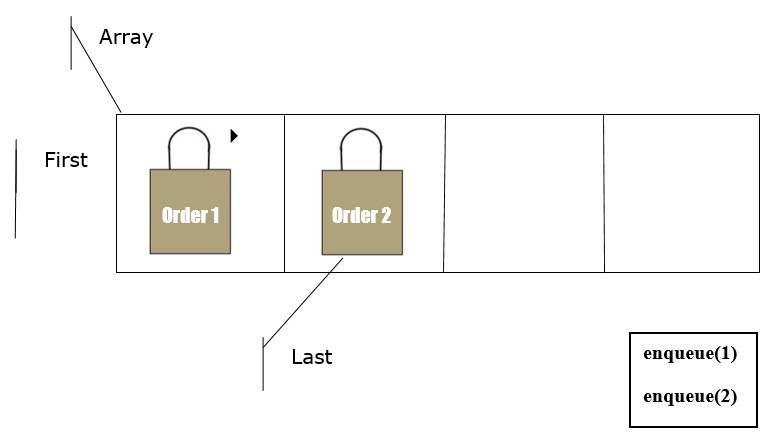
**[1]**



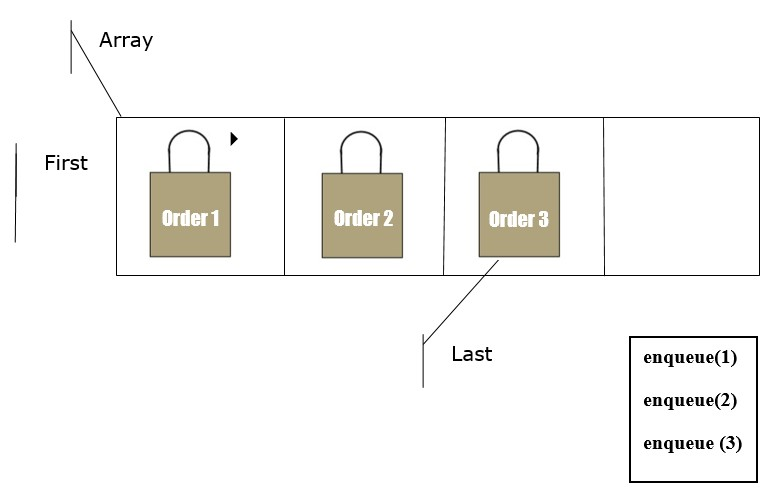
**[2]**



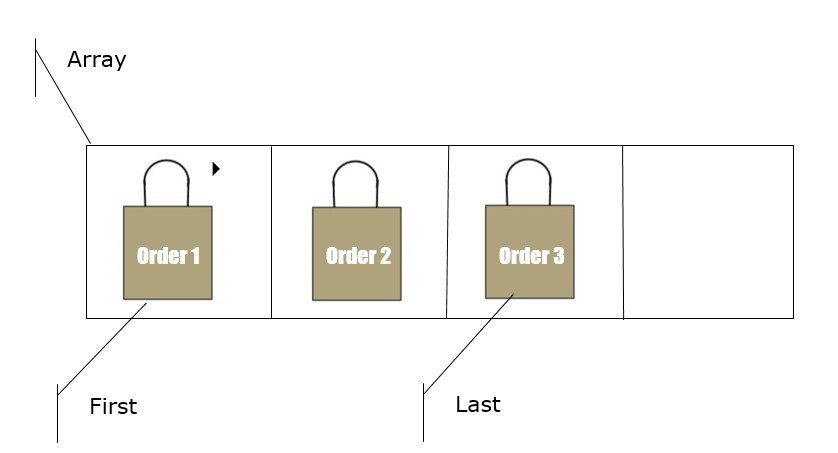
**[3]**



**[4]**



**[5]**



**[6]**

**A queue is a first-in-first-out (FIFO)** data structure. Which means that the first element added to the queue will be the first element removed.

Which can use a queue to handle requests in first order - first order handler by adding new orders to the end of the queue and processing requests at the front of the queue.

**Here is a brief explanation of how this works in briefly: [1]**

Ordering 1🡺 Ordering 2 🡺 Ordering 3

In this example, order 1 is added to the queue first, followed by order 2, order 3. When it comes time to process orders, the first order in the queue (order 1) is removed and processed. The next order in the queue (order 2) becomes the first in the queue, and so on.

This ensures that requests are processed in the order in which they are received, and that no command is skipped or processed out of sequence. The queue acts as a buffer that holds orders until they can be processed, allowing the system to efficiently handle a potentially large number of requests.

**So, you will explain step by step:**

**1-** We will call the first order that entered the drive-through: first.

**2 -** We will name the last order on the drive-thru: last.

**Note:** If another order enters the "drive-thru service", it will be the last order, and so on.

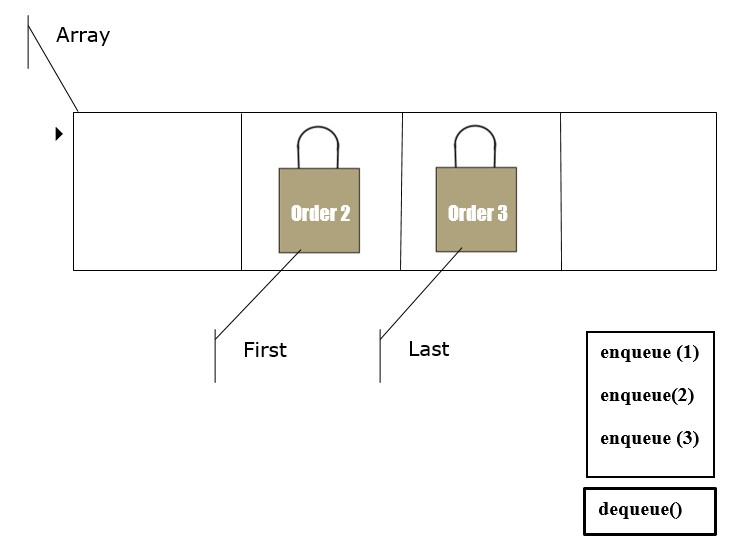
Therefore, we discover that the process of adding "enqueue" is on the side of "Last" and also when the process of removing "dequeue" is on the side of "First". **[1]**

Suppose we have an "Array" of 4 orders and we add the order 1 and the process of adding in Queue is called "enqueue".

Therefore, we need a variable in the addition process, which is Last, and we also need a variable in the deletion process, which is First. **[2]**

And because the queue is open on both sides, which is from the side of Last and from the side of First, to which the queue is applied:

* And when adding order 1 "enqueue (1)", the Last process will move and stop at order 1. **[3]**
* So, let's add another order 2 "enqueue (2)”, Last will move and stop at order 2 in Array. **[4]**
* And again, if we add order 3, "enqueue (3)" "Last" will move and stop at order 3 in the "array". **[5]**
* So, who is the order that came last in the "Last" queue? it is order 3, so if we ask ourselves who is the first in this queue? Definitely order 1. **[6]**



Created by me.

* And if we want to do the removal process, then in the process of removing and applying the idea of the queue, order 1, “dequeue ()”, will be deleted, because it is the first element to enter the queue, and this achieves the principle of first-in-first-out, so after the removal process, the "First" order will be order 2, and if we ask ourselves, can we delete order 3 or order 2 before we remove element 2? And the answer here, of course, is “no”, because since we are dealing with the queue, when we do the removal process, we cannot do the removal process, except in the first element that entered this queue, and in the process of adding, we will add after the last element that was added to this queue.
* We know that when the queue is empty, the "first" and "last" pointers are usually set to -1. This indicates that there are no orders in the waiting list.
* The "first" index indicates the index of the first vehicle to be removed, while the "last" index indicates the index where the next order will be inserted.
* Setting the initial value to -1 provides a clear indication that the queue is empty. It allows simple conditional checks to determine if the queue is empty. For example, if "First" and "Last" are both -1, it means that the queue has no orders and can be easily handled in conditional statements.

**1.5:** Consider implementing the Queue class over an array, and SLL. **Discuss** the implementation details in term of space and time complexities as well as the possible trade-offs regarding queue specifications (enqueue and dequeue).

**Answer:**

**First, the queue over the array:**

In the queue, which we mentioned in the previous question, which represents the First-In-First-Out (FIFO) data structure, which means that the first element to be inserted is the first element to be removed, so we will explain a comparison of the three methods of queuing across the array:

**First approach:**

First, the constructor, the Queue class initializes the array "arr" with default size 5 and sets the first and last indices to -1, and if the time complexity is O (1), because it takes a constant initialization time, and the space complexity is O(n), because n is the size of matrix. And in this case. n is set to the default size of 5.

Second, isEmpty (), which this method uses to check whether the queue is empty by comparing whether the first and last pointers are -1. Also, the time complexity is O (1), because it performs fixed time operations. Third, isFull (), which checks whether the queue is full by comparing whether the last pointer is of size -1, which also has a complication. O (1) because it performs fixed time operations. Fourth, resize () This method is executed when the queue is full, and its purpose is to double the size of the queue. Where you create a new array of double size, copy the elements from the old array into the new array, and then replace the old array with the new one. So, the time complexity is O(n) because it requires copying n elements from the old array to the new array, and the space complexity is O(n) because it creates a new array of size n.

Fifth, the queue (int v) that implements or uses this method to insert an element at the end of the queue. If the queue is full, it calls the resize () method to double the size of the array. Then it increments the last pointer and inserts the new element at the last position. If the queue is empty before the operation, it also increments the first pointer. The time complexity is O (1) because it generally requires a fixed number of operations, but in the worst-case scenario when the queue is full, the time complexity is O(n) due to the resize () method. Finally, the spatial complexity is O (1) because it does not require any additional space to insert an element. Sixth, dequeue (), which removes an element from the front of the queue. Also, it first checks if the queue is empty, and if it is empty, it returns -1. Then, it retrieves the first element, and if that is the last element in the queue, it resets the first and last element to -1. If there are more items, they increase first. The time complexity is O (1) because it performs a fixed number of operations, and the space complexity is O (1) because it does not require additional space to remove an element.

**The second approach is shifting array:**

Which will be the modification in the dequeue () operation, which is the idea here and in this approach, that the dequeue () method not only removes the front element, but also moves all the remaining elements one place to the left. If the value stripped from the class is the last, then both the first and the last value are reset to -1. Otherwise, the loop moves all elements from the first + 1 to the last one place to the left, and the last goes down. The time complexity of this method is O(n) due to the transformation of the elements, and the space complexity is O (1).

So, the second approach implements a queue using an array of switching elements on desorialization. This ensures that the front of the queue always remains at the zero index of the array. However, this comes at the cost of increasing the time complexity of the dequeue () operation, which becomes O(n) because all elements need to be converted. Thus, the choice between the two approaches will depend on the specific use case and whether faster queuing operations or removing rows is more important.

**The third approach, Circular Array:**

Here the modification will be in the isfull() method for the first approach

First, isfull (), which this method checks if the queue is full by checking two cases: (1) if the last is at the end of the array and the first is at the beginning, and (2) if the first precedes the last position. Both checks have a time complexity of O (1), which is because they only require constant time operations. Third, resize () which doubles the size of the array when the queue is full. The elements of the new array are filled in two parts if the queue is circular, that is, the first is not at index 0. The time complexity of this method is O(n), where n is the number of elements in the queue because all n elements are copied to the new array. The space complexity is also O(n) as a new array of multiple size is created. Fourthly, enqueue (int v) This method adds a new element to the queue. If the queue is full, it changes the size of the array. Next, it checks whether the queue is empty, and if it is empty, it will be incremented first. If the latter is at the end of the array and the first is not at the beginning, the latter is wrapped around the beginning of the array. Else, the latter is increased. Then the new element is inserted in the last position. The average time complexity is O (1) because it usually requires a fixed amount of operations. However, the worst time complexity is O(n), when resizing is required. The space complexity is O (1).

Finally, the dequeue () method will be like the first approach, which is dequeue () which removes an element from the front of the queue. If the queue is empty, it returns -1. The element is initially stored in dequedVal to be returned later. If the stripped element is the last element, first and last are reset to -1. Else, the increment is done first, possibly wrapping around the beginning of the array. The time complexity is O (1) because it only requires a fixed amount of operations, and the space complexity is O (1) because it doesn't require any extra space to remove an element.

So, with three approaches, this approach offers an advantage over the first two because it makes efficient use of the array by treating it as a circular buffer. This eliminates the need to shift elements in the array when an element is detached, as in the second method, and it also manages the "blank space" at the beginning of the array better than the first method. Queuing and dequeuing operations are still effective, and array resizing remains the most expensive operation, but only occurs when the queue is full.

|  |  |  |  |
| --- | --- | --- | --- |
| Approaches’ | Approach 1: Queue Array | Approach 2: Shifting Array | Approach 3: Circular Array |
| Enqueue Complexity -avg | O (1) | O (1) | O (1) |
| Enqueue Complexity -worst | O (n) | O (n) | O (n) |
| Dequeue Complexity | O (1) | O(n) | O (1) |
| Resize Complexity | O (n) | O (n) | O (n) |
| Space Complexity | Wasteful | Full usage | Optimal |
| Implementation Complexity | Low | Medium | Medium-High |

Created by me.

**Secondly, Queue over SLL:**

isEmpty () function: This method checks whether the linked list is empty by checking if the head is null. The time complexity is O (1) and space complexity is also O (1), as this operation only involves one comparison and does not use additional memory space.

addToHead (int v) function: This method adds a new node at the beginning of the linked list. The time complexity is O (1) because it directly accesses the head of the list and does not traverse any elements. The space complexity is also O (1), as it only creates one new node irrespective of the size of the linked list.

addtoTail (int v) function: This method adds a new node at the end of the linked list. The time complexity is O(n) because it needs to traverse the entire list to find the tail. The space complexity is O (1) as it only creates a new node.

removeFromEnd () function: This method removes a node from the end of the linked list. The time complexity is O(n) because it needs to traverse the entire list to find the tail. The space complexity is O (1) as it only uses a few variables to hold values and references temporarily.

removeFromHead () function: This method removes a node from the beginning of the linked list. The time complexity is O (1) because it directly accesses and modifies the head of the list. The space complexity is also O (1) because it uses a few temporary variables to hold data and references.

printAll () function: This method prints all the nodes in the linked list. The time complexity is O(n) because it needs to traverse the entire list. The space complexity is O (1) because it only uses one node to traverse the list and print values.

getTail () function: This method returns the tail of the linked list. The time complexity is O(n) because it needs to traverse the entire list to find the tail. The space complexity is O (1) because it uses one node to traverse the list.

The Queue class: enqueue (int v) function: This method adds an element at the beginning of the queue. This operation requires adding an element to the head of the linked list, which takes O (1) time complexity. The space complexity is O (1) because it creates a new node for each enqueue operation.

dequeue () function: This method removes an element from the end of the queue. This operation requires removing an element from the end of the linked list, which takes O(n) time complexity. The space complexity is O (1) because it uses a few variables to hold data and references temporarily. The time complexity could be improved to O(1) if we maintain a tail pointer in the SLL.

isEmpty (): As the time and space complexity is O (1), this operation is extremely efficient. The trade-off here is minimal. In exchange for this efficiency, we assume that the list's head is maintained correctly, i.e., it should point to the first node or be null if the list is empty.

addToHead (int v): Again, with time and space complexity of O (1), the trade-off is minimal. The assumption here is that adding to the start of the list will not disturb the order of existing elements, which is a constraint if maintaining order is crucial.

addtoTail (int v): The time complexity is O(n), which can be costly for large lists, as it has to traverse the entire list. However, this ensures that the order of elements is maintained. Space complexity is O (1), which is efficient.

removeFromEnd (): This function also has a time complexity of O(n) due to list traversal, which may not be ideal for large lists. However, it offers the flexibility of removing elements from the end while maintaining the order of the remaining elements. Its space complexity is O (1).

removeFromHead (): Time complexity is O (1), which is very efficient. But this function only allows removal of elements from the beginning of the list. If there's a need to remove an element from a different position, this function can't be used. Space complexity is O (1).

printAll (): This function has a time complexity of O(n), which is less efficient for large lists as it requires traversing the entire list. The trade-off is that it provides a way to visualize the entire contents of the list. Space complexity is O (1), as it only uses a temporary node.

getTail (): This function has a time complexity of O(n) as it needs to traverse the entire list. The trade-off is it provides access to the last node of the list. Space complexity is O (1).

Regarding the Queue class:

enqueue (int v): This operation has a time complexity of O (1), which is highly efficient for adding elements. But it only allows adding elements at the beginning, which might not suit all use cases. Its space complexity is also O (1), as it only creates a new node.

dequeue (): This operation has a time complexity of O(n), which may not be efficient for large lists as it requires traversing the entire list. The trade-off here is that it allows removal of elements from the end, which is characteristic of a queue. As you suggested, the time complexity can be improved to O (1) if a tail pointer is maintained. Its space complexity is O (1).

**1.6:** In Mac they have a job to put plates in a cabinet. The way they put the plates is one on top of another until the cabinet is full. When they want to use the plates, they take the plate that is on top in the cabinet until they are all finished. They have brought a robot to do the job. Help them by **defining** a stack specification so they can use it to implement the solution, **show** how stack handle the solution.

**Answer:**

a stack is a data structure that follows the Last-In-First-Out (LIFO) principle. This means that the last element to be inserted or added to the stack is the first element to be deleted or removed from it, and this corresponds to the concept of the described scenario: in which the plates are stacked one on top of the other (Last-In) and the top plate is the first-out plate (First-Out).

**Stack Specification:**

We define the stack specifications, which will be the locker where the panels are stored. The stack may contain a fixed number of panels, in what is meant by "the maximum capacity of the cabinet.

Which has two main operations, 1- pushing and 2- popping. The push operation adds an element to the top of the stack, and the pop operation removes the topmost element from the stack. Which follows the principle of Last-In-First-Out.

So, Implementing the using Stack Operations:

First, we create an empty stack with a capacity commensurate with the maximum capacity of the cabinet. When a new board arrives, the robot pushes to add the board to the top of the stack. This process mimics placing the painting on top of a stack of paintings in a closet.

On the other hand, if the cabinet is already full and there is no space to add more panels, then the push operation will fail or result in an error. This indicates that the treasury is at its maximum capacity.

When we help McDonald's when an employee needs a plate, the robot pops to remove the top plate from the stack. This process simulates taking the painting from the top of the pile in the closet.

If there are no more plates in the stack (that is, the stack is empty), then the pop operation will fail or produce an error. This indicates that there are no plates left in the cabinet.

Thus, the robot continues the push and pop operations when the boards arrive or are needed, respectively, until all the boards are used.

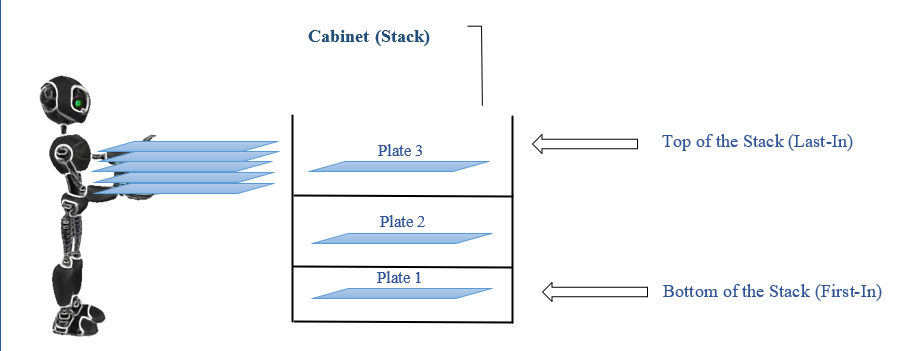
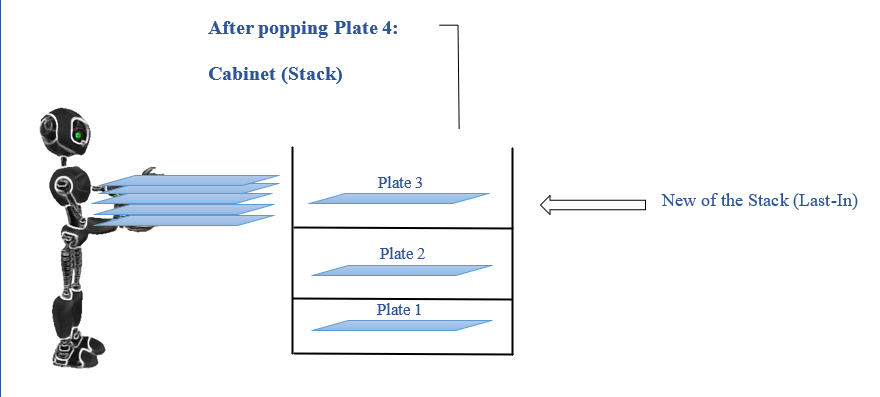
**If we ask ourselves, how can the stack help solve this problem:**

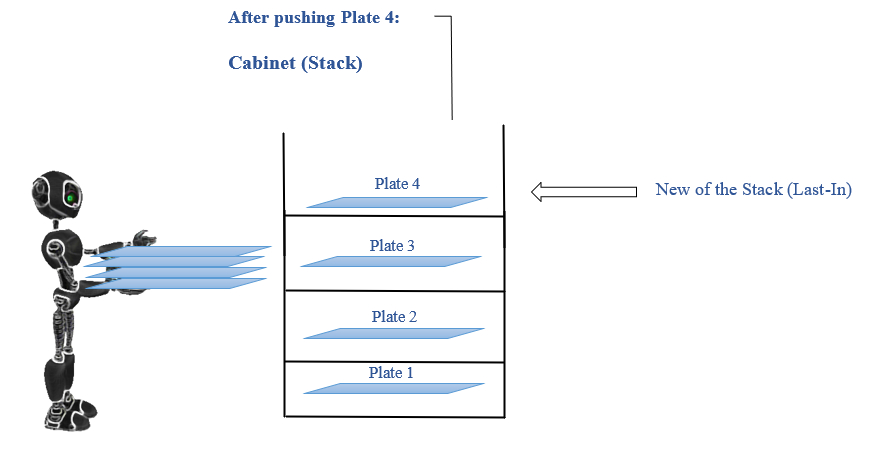
When the stack is used, the bot ensures that the last board added (which is the most recent board) is the first board to be retrieved. This complies with the requirement to take the board up when needed, according to the LIFO principle. The push-stack operation allows the robot to add plates on top of each other, simulating how the plates are stacked in a cupboard.

Which enables the stack-popping operation of the robot to easily retrieve the top plate, thus repeating the procedure of taking the plate from the top of the stack into the cabinet.

This ensures that the stack capacity constraint (which is proportional to the maximum capacity of the locker) ensures that the robot does not exceed the space available in the locker and handles the condition when the locker is full.

Finally, by using stack specifications and using push and pop operations, the robot can effectively and efficiently handle the task of organizing the panels in the cupboard. Which mirrors the LIFO behavior and operations of the stack, providing an efficient and logical solution to the problem.





**[2]**

**[1]**

**[3]**

Note: Pictures summarize the explanation and show the processes of pushing and popping.

Created by me.

**Task 2: ADT and OOP**

**2.1:** You have implemented the above solution using the Object-Oriented Programming principles. **Negotiate** the idea that ADTs are the basis of object orientation. Justify your answer whether you agree or disagree.

**Answer:**

The idea that abstract data types (ADTs) can be considered as the basis for object-oriented programming (OOP). ADTs are data structures that are defined by their behavior (that is, the operations that can be performed on them) rather than by their execution. This illustrates and sustains a core idea in OOP - that objects are defined by their interface, the methods they provide, rather than their implementation - which ADTs look at or support, and which is sometimes seen as the theoretical basis for OOP.

Also, ADTs are a basic and important idea in data structures because they enable and provide a way to encapsulate data and associated procedures, which encourage abstraction and information hiding. In addition, they specify the type of data and the operations that can be performed on it, but they do not specify how these procedures are performed.

In the provided code, we can see examples of dynamic helpers such as the 'item' and 'request' classes, which are defined by their attributes and methods (locators and selectors) rather than their implementation details. The "Queue" class is also an example of an ADT, being defined by its behavior (enqueue- addOrder, dequeue -removeOrder, isEmpty) rather than its implementation details.

In addition, we can also see the use of encapsulation, which is another basic concept in OOP. Encapsulation refers to the practice of keeping an object's internal state and behavior hidden from the outside world, and only reveals a public interface for interacting with the object. In the provided code, the properties of the Client and Request classes are declared and can only be accessed or modified by their corresponding public methods.

**Which defines and extends the concept of OOP with several principles:**

Inheritance: This principle allows for hierarchy and reuse. Which a new class can inherit from members (properties and methods) of an existing class, and extend or override them as necessary.

Polymorphism: which defines and allows the use of a single interface with different basic forms of data. For example, method overloading and method overriding.

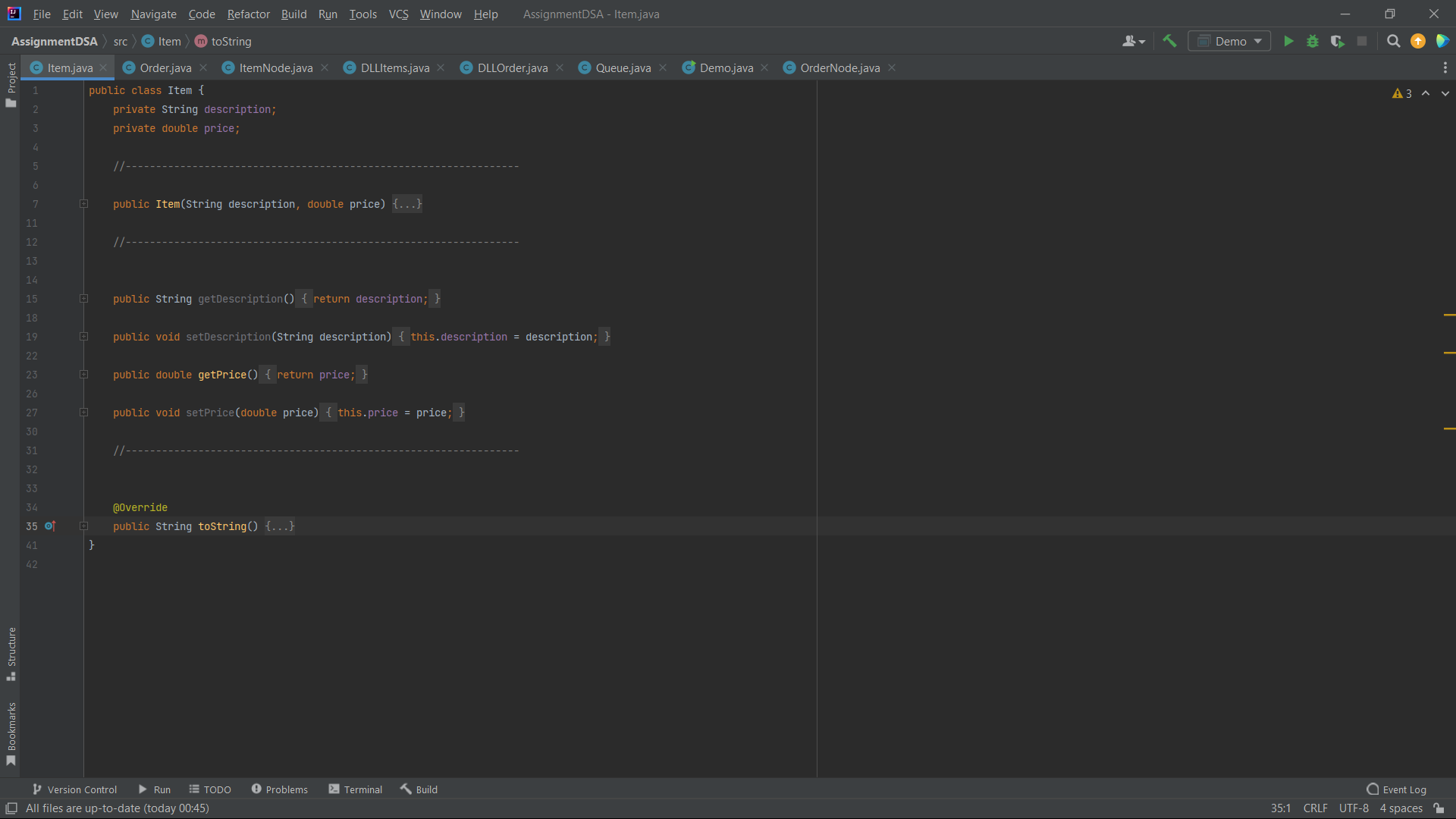
Abstraction: which is the idea of dealing with complex systems by managing different levels of abstraction. Abstract classes and interfaces in Java help achieve abstraction.

You've created classes for Item, Order, ItemNode, DLLItems, OrderNode, DLLOrder, and Queue - each of which can be viewed as an implementation of ADT, because they encapsulate related data and operations.

In this sense, one could argue that OOP is an extension of the basic principles of ADTs, and provides additional mechanisms (such as inheritance, polymorphism, and encapsulation) that make it easier to build complex systems. On the other hand, although all default processing tools use the principles of ADT tools, not all uses of ADT tools are necessarily object-oriented. So, for example, ADT tools can be implemented in procedural languages like C, which do not support object-oriented features.

In general, yes, I agree with the statement that ADTs form the foundations of OOP. The code provided demonstrates the use of ADTs and encapsulation, which are basic concepts in OOP. These concepts are also clearly implemented, and it is easy to understand what is happening in the code, which makes it a good example of OOP. However, OOP expands on these principles and additional features that facilitate the development of more complex systems.

**For example:**

****

In this class, the description and price attributes are marked as private, which means they cannot be accessed directly from outside the Item class. However, they can still be accessed and modified indirectly through the public getDescription, setDescription, getPrice, and setPrice methods. These methods are the only way to access and modify the description and price attributes, ensuring that the Item class has full control over how its data is used and manipulated.

This kind of encapsulation and data hiding helps to ensure the integrity of the data within the class, as it prevents the data from being accessed or modified directly from outside the class. It also provides the flexibility to change the class implementation without affecting other parts of the program that use the class.

**2.2:**  **Explore** the advantages of encapsulation and information hiding, class diagram representation when using the queue ADT.

**Answer:**

My implementation is a good example of how object-oriented principles like encapsulation and steganography can be used in practice. Here's how to apply these principles:

* **Encapsulation and Information hiding (Data Hiding):**

In the Item, Order, DLLItems, DLLOrder, and Queue classes, variables are declared as private, which means that they cannot be accessed directly from outside the class. Instead, the getter and setter methods are used to retrieve or modify the values of these variables. Which illustrates this as a clear example of encapsulation and steganography.

Also, for example, consider the Queue class: the DLLOrder instance List and the first and last OrderNode instances are private attributes. It cannot be directly accessed or manipulated. Instead, we use the enqueue, dequeue, and isEmpty public methods to interact with the queue.

**Encapsulation** in Java refers to a code-wrapping mechanism that binds the code and the data it processes together into a single unit, for example a medicine capsule has mixed types of modules inside. We can use getter and setter methods to specify the data and how to get it.

Encapsulation kind of sounds like in a little capsule so if you think of like a pill everything is inside a pill.

But during capsulation in java all the setting of variables of a class is inside a method.

**Private:** The level of access to the private modifier will only be within the class itself and cannot be accessed from outside the class, this means that you can see it in the same class, but if it is outside the class, you will not be allowed to see it. For Example:

In the Order class, the attributes orderID, listOfItems, and totalPrice are all declared as private. This means that these attributes can't be directly accessed or modified from outside the class. Instead, public getter and setter methods are provided to interact with these private attributes.

public class Order {

private int orderID;

private DLLItems listOfItems;

private float totalPrice;

...

**Public:** If we set the global modifier access level to be public, this means that we can access it from inside the class, from outside the class, from inside the package, and from outside the package.

In the Item class, all of the methods are public, including the constructor, getters and setters, and the toString method. This means that these methods can be called on an Item object from anywhere. For Example:

public class Item {

public Item(String description, double price) {...}

public String getDescription() {...}

public void setDescription(String description) {...}

public double getPrice() {...}

public void setPrice(double price) {...}

public String toString() {...}

}

**Protected:** Here is the protected access level

Rate inside and outside the package through sup class. If we don't sup class, not accessible from out of the package. This means that I can see the attributes or do an access operation from the super class in addition to the sup class. In this case, the Access Modifiers are protected, meaning that any methods or attributes are protected. Direct access from the super class will work for them, in addition to the sup class.

In the protected, any class can be accessed if it is inside the package, but if the class is outside the package, it cannot be accessed. For Example:

public class ItemNode {

protected Item value;

protected ItemNode next;

protected ItemNode prev;

//...

}

In this class, value, next, and prev are defined as protected. This means they can be accessed directly from within the same class, any classes in the same package, and any subclasses, even if they are in different packages.

**Advantages of using encapsulation and Information hiding (Data hiding) in this context include the following:**

* First, data integrity: By controlling the ways in which data or information can be accessed or modified (for example, not allowing negative prices on an item), we can maintain the integrity of the data.
* Second, flexibility and maintainability: Encapsulation allows the internal implementation of a class to be changed without affecting the parts of the program that use the class. This makes it more flexible and easier to update or modify your code in the future.
* Third, controlled access: Steganography prevents unauthorized operations on data.
* Fourth, simplify use: ADT queue users do not need to understand and know its internal implementation. They interact with it through a standard set of operations such as enqueue, dequeue, isEmpty, and so on.
* Security: Encapsulation provides a way to protect data that may be inadvertently accessed or changed. By making data private within a class, we can control who can change it and how.
* **Class Diagram Representation (UML):**

Which class schema of this code includes classes such as Item, Order, ItemNode, DLLItems, DLLOrder, OrderNode, Queue, Demo. The relationships, features and methods of these categories can be represented graphically to give a clear and understandable picture and structure of the system as a whole.

For example, Order is composed of DLL elements and is associated with an OrderNode in the DLLOrder class, which has a composition relationship. which DLLOrder, in turn, is used in the Queue class, and they all interact in the Demo class.

**The advantages of using class diagrams in this context therefore include:**

* First, better understanding: Class diagrams provide a clear and understandable visual representation of how the system works, making it easier for developers and stakeholders to understand and speed up implementation.
* Second, effective communication: Blueprints are an excellent tool for communication, allowing developers to convey about how the system is working efficiently and effectively to other stakeholders or developers.
* Third, usability: The class diagram also helps to understand the usage of different classes and understand the relationships between classes, along with their relationships, and enhance usability in the design and coding stages.
* And fourth, documentation: Class diagrams provide an important foundation in system documentation that helps team members and new developers quickly understand the system.

**2.2:** **Assess** three benefits of implementing independent Data Structure such as queue.

**Answer:**

So, with regard to the implementation of independent data structures, such as the queue, it has many advantages. Therefore, I will present three main advantages, and we will evaluate and explain them in detail:

**First, encapsulation and data integrity:** A queue, which as an independent data structure, which abstracts all its data and operations. What this means is that it is not possible to access the data in the queue directly, and instead we have to use specific methods provided by the queue, such as enqueue, dequeue, and isEmpty, which ensures the integrity of the data and ensures that the user is prevented from accessing other data, as the internal state of the queue is protected from inappropriate change or modification. Encapsulation allows us to control how data is accessed or processed, thus preventing misuse and facilitating ease of use and understanding.

For example, every new request is added at the end of the enqueue, and each request is submitted from the front of the queue (dequeue), thus we guarantee the "first come, first served" principle, and by implementing the emission list as an independent data structure, we encapsulate all the details necessary to manage the arrangement of these transactions. That is, only specific methods such as enqueue (adding new orders) and dequeue (completed orders) are displayed, while private data and basic operations are preserved. This ensures data integrity and prevents misuse, such as serving orders out of turn.

**Second, Modularity and Code Reusability:** Implementing the queue as an independent data structure promotes modularity and reusability, which can be used as a good queue in executing any program that needs to maintain the FIFO command or principle without having to know the details of how to implement the data. Waiting list. So, it's a straightforward case of writing once, using it many times, which saves flexibility, reduces redundancy, and increases maintainability and code scalability.

For example, a queue can also be used inside a restaurant to sit down, which can use a queue to manage tables. So, when clients arrive, they are added to the queue. When tables become available, the host can take the first group from the queue and sit on it. This ensures that the tables are populated in the order that the collections arrive in and that the process is fair for all clients, so we can use the same queue data structure to handle these types of things, this way we don't have to rewrite the same logic and data from scratch again, which saves time and effort and makes the code maintenance process easier, more efficient and effective.

**Third, abstraction and simplification of complex problems:** which removes independent data structures such as queues and the complexity of specific and specific operations, allowing developers and programmers to solve complex problems more simply and more efficiently. Queue which is a perfect illustration of abstract data types - ADT which presents a clear concept of queue from real life. And that when a queue is used, developers can focus on what it does (e.g., sticking to a FIFO order), rather than how it does it.

In the McDonald's app, you may have a feature which is dealing with the preparation of different food items. Which different food items take different lengths of time to prepare and it is necessary and important to manage them in an efficient way to ensure prompt service.

So here, the queue can be a valuable and important tool. Which every new food item prepared can be added to the queue as soon as it is ordered. So, the kitchen staff then needs to always work on the item at the front of the queue, and remove it once it's done (dequeue). That way, you don't have to worry about which item to prepare next - the queue takes care of that. This abstraction allows developers to focus on the broader logic of the application, while the queue takes care of managing the order of food preparation.

**Task 3: Testing sorting algorithms to improve orders handling**

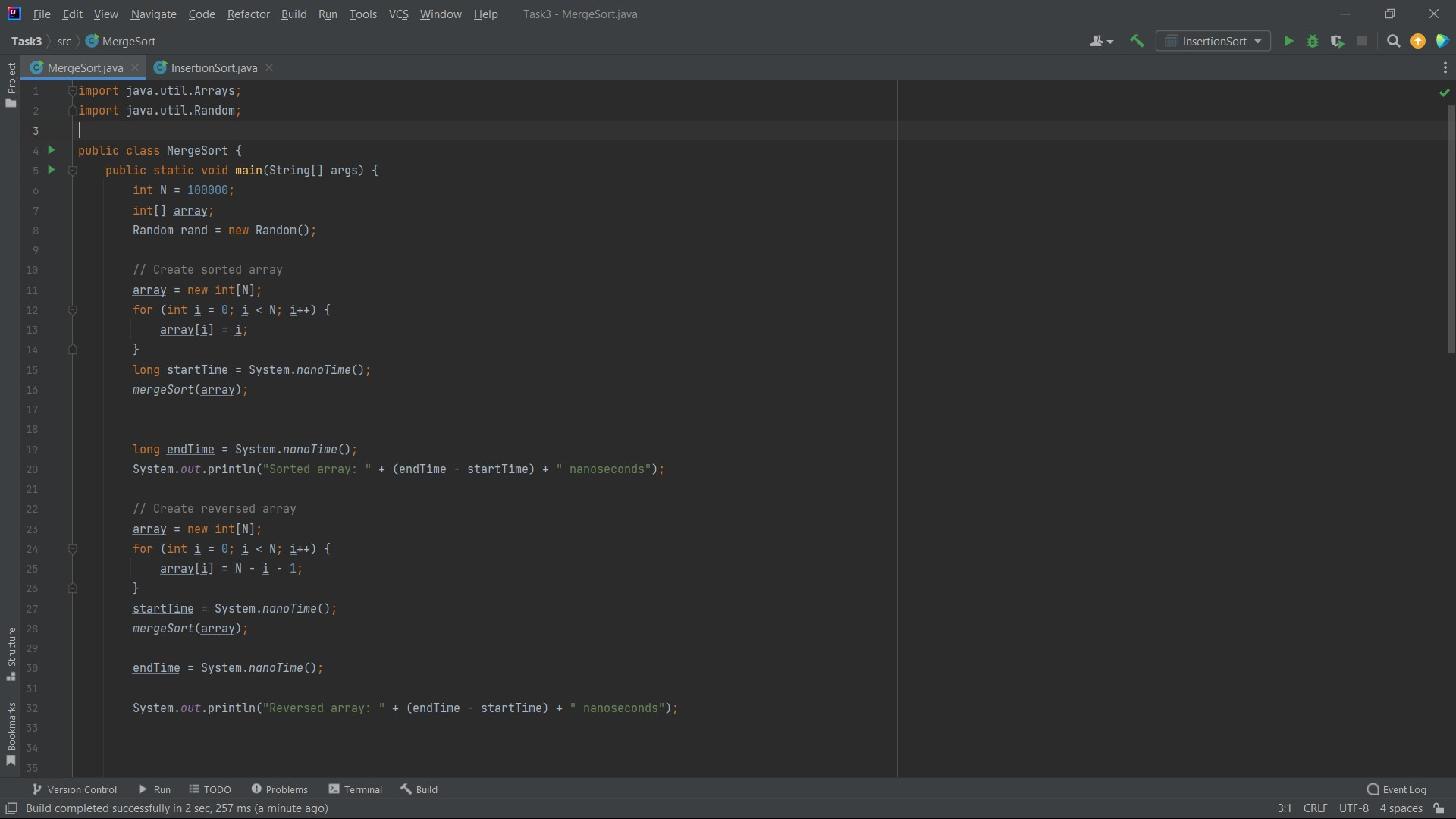
You have been asked to implement a feature to sort the orders before adding them to the queue based in their total price. This will give orders with highest price the priority to be handled first. For the easiness of implementation and demonstration, assume that you are given an array of total prices. Sort it in **descending** order using two sorting algorithms (merge sort and insertion sort). And **compare** their performances using time benchmarking technique over three different shapes of arrays.

Use the following table to organize your results, N is array size:

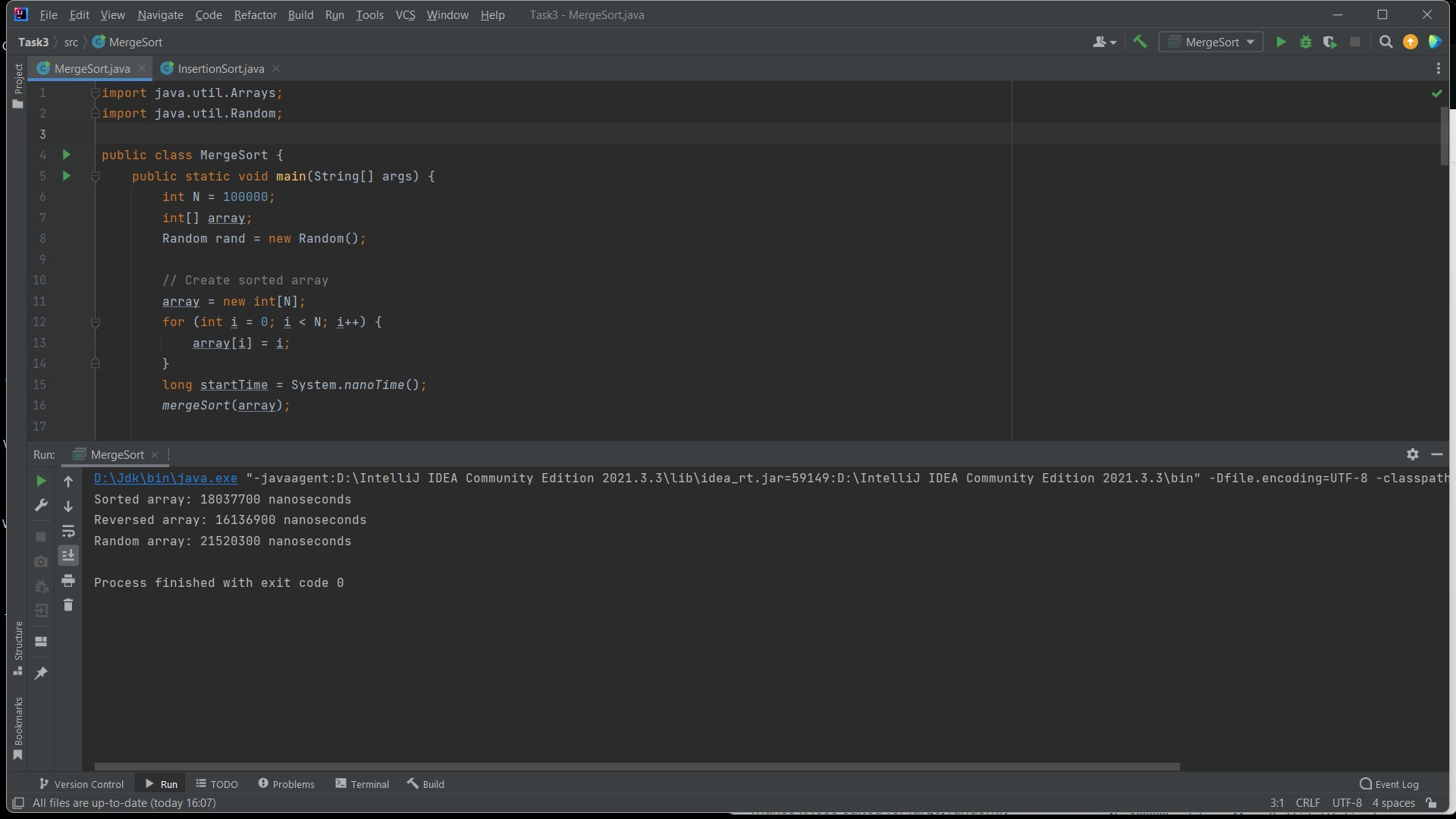
Answer:

|  |  |  |  |
| --- | --- | --- | --- |
| **N= 100000** | **Sorted** | **Reversely sorted** | **Random** |
| **Merge sort** | **18037700 nanoseconds** | **16136900 nanoseconds** | **21520300 nanoseconds** |
| **Insertion sort** | **1884323100 nanoseconds** | **1409200 nanoseconds** | **773637700 nanoseconds** |

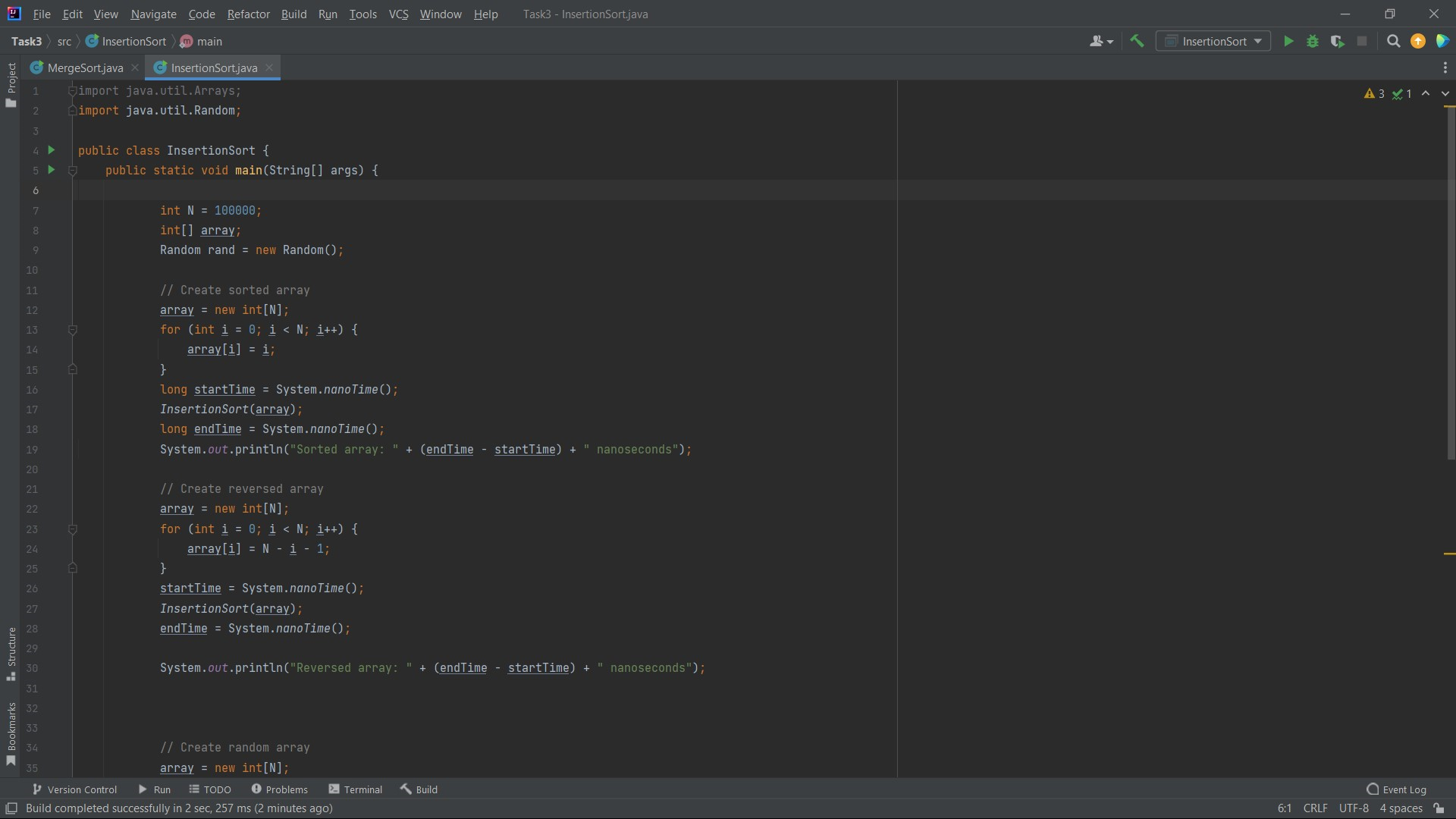
**- Class MergeSort:**



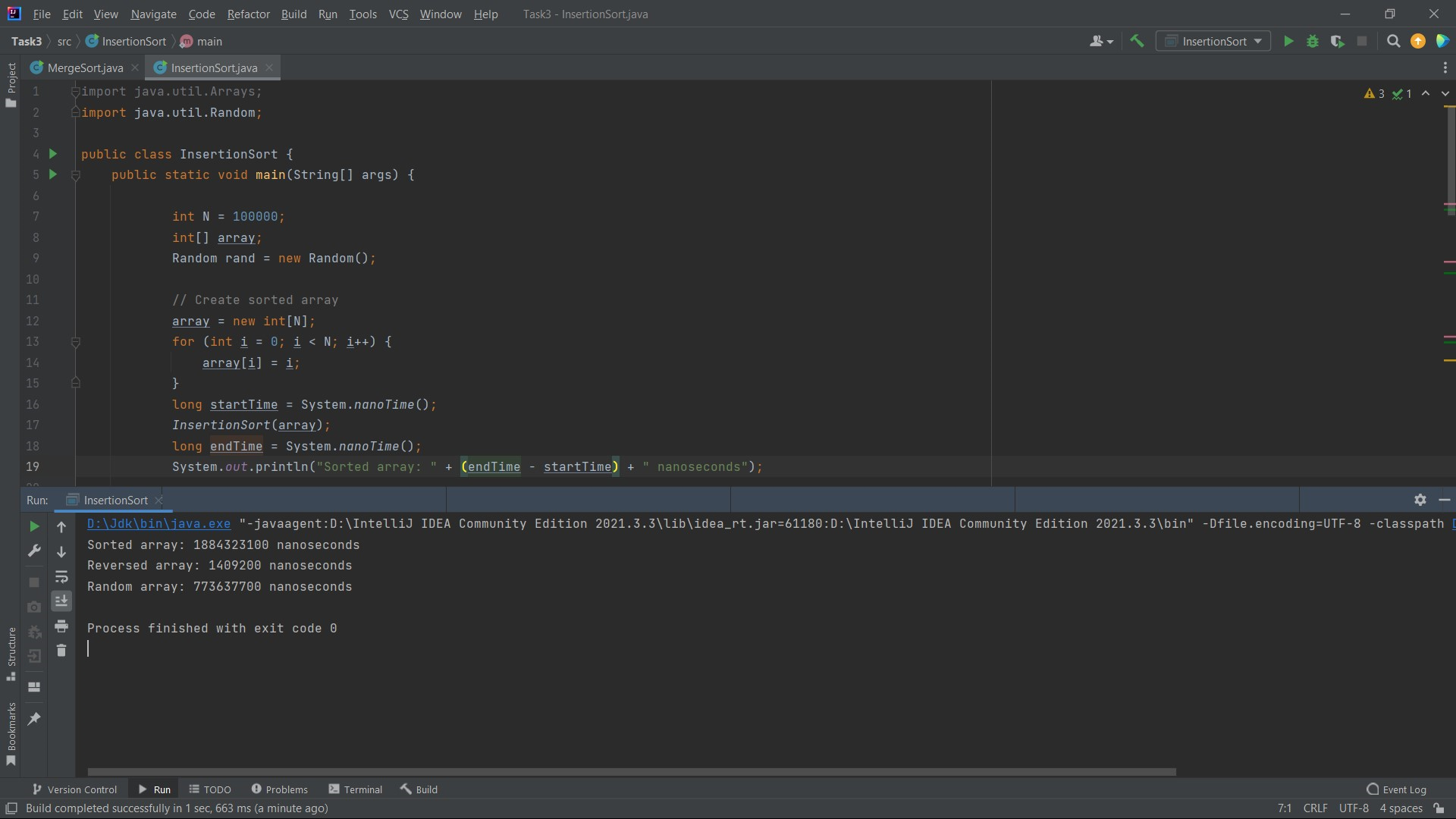
**- Output:**



**- Class InsertionSort:**



**- Output:**



**3.1:** Consider how asymptotic analysis is used to evaluate the effectiveness of algorithms. Use the sorting techniques used as an example.

Answer:

**Asymptotic analysis:**

Asymptotic analysis is a method for describing restricted behaviour. The term convergence means to approach a value or curve arbitrarily (i.e., as some kind of limit is taken). In the context of algorithms, asymptotic analysis is a method of describing their performance in terms of execution time as a function of input size. It provides an upper bound on the growth rate, which helps us predict how runtime or space requirements will grow as the size of the entry grows.

As you mentioned, it's input bound: without any input, the algorithm will have a constant time complexity of o (1), because it doesn't perform any operations based on the input.

The three common cases we discuss in asymptotic analysis are:

best case: this is the scenario in which the algorithm performs best. For example, in a search problem, the best case occurs if the target element is the first in the list.

intermediate case: this is basically expected performance over all possible inputs, assuming all inputs are equally likely. The calculation is often more difficult because you need to consider all possible inputs and their probabilities.

worst case: this is the scenario in which the algorithm performs worst. For example, in the same search problem, the worst case occurs if the target element is at the end of the list or is not there at all.

Asymptotic analysis is like taking a snapshot of an algorithm's behaviour as we keep on increasing the input size. Picture it like watching a runner in a marathon - you want to see how their speed changes as the race progresses. In this case, the runner is the algorithm and the race is the computation it needs to perform.

The term 'convergence' here is kind of like saying a runner is getting closer and closer to the finish line. In the world of algorithms, it's about how their performance 'converges' to a certain pattern as the input size gets larger and larger.

But what's really neat about this is that it gives us an upper limit, or a worst-case scenario of how the algorithm will perform. It's kind of like knowing the slowest speed our runner will go. This can be super helpful for understanding how the time or space an algorithm needs will grow as we feed it more data.

**Big O notation:**

Big O notation, denoted by O, is a mathematical notation that describes an upper bound on the time complexity in a worst-case scenario. It provides an upper bound on the time complexity, allowing us to say that the time complexity is "at most" a given value. This gives us an upper bound for the algorithm, which can be useful when comparing two algorithms.

For example, if we say that an algorithm is o(n^2), then we say that in the worst-case scenario, the time complexity of the algorithm will increase quadratic with the size of the input. This does not tell us the exact number of steps the algorithm will perform, but it does give us an idea of how the algorithm will scale.

In essence, asymptotic analysis and big o coding provide a high-level means of comparing algorithm performance by summarizing system details (such as hardware or programming language), focusing only on the way performance grows with respect to input size.

Asymptotic analysis, specifically Big O notation, is a mathematical notation that describes the finite behaviour of a function when the argument tends towards a certain value or infinity. In computer science, this concept is widely used to describe the performance or complexity of algorithms.

When analyzing algorithms, it is important to understand how the algorithm's performance scales as the input size increases. This is especially important for large data sets where algorithm performance can greatly affect the overall efficiency of the system.

The main idea of asymptotic analysis is to obtain a measure of the efficiency of the algorithms that does not depend on specific machine constants, and does not require the implementation of the algorithms and measurement of actual times. It allows us to rank algorithms in terms of their growth rates as the input volume increases.

The Big O notation represents the upper bound of the time complexity in the worst-case scenario. Provides an upper limit on the time the algorithm takes in terms of input size (n).

Thus, it provides a worst-case scenario for the algorithm, ensuring that the time complexity does not exceed a certain time for a given input size.

In coding, we usually drop constants and lower order terms. This is because, as the input size increases, the constants and lower-order terms become less important in the overall computation time. It will be the higher order terms that most affect the time complexity. For example, if we have a time complexity of 3n^2 + 4n + 5, it will be represented in Big O notation as O(n^2).

Common Big O time complexities, ranked from best to worst, include: O (1) - constant time, O (log n) - logarithmic time, O(n) - linear time, O (n log n) - linear log time, O(n^2) - squared time, O(n^3) - cubic time, and O(2^n) - exponential time.

Big O notation is also useful for analyzing the spatial complexity of an algorithm. Just like time complexity, space complexity provides an upper limit on the memory required by the algorithm in terms of input size. For example, an algorithm that sorts an array in place (that is, it does not create any additional data structures) could have a space complexity of O (1), while an algorithm that creates a new array of size n could have a space complexity of O(n).

In addition to the time and space complexity, Big O notation can help identify shortcomings in the algorithm. By understanding the algorithm's Big O notation, we can determine whether the algorithm will scale with larger input sizes, become too slow, or use too much memory.

It is important to understand the difference between worst time complexity and average case. The worst-case time complexity is the maximum time the algorithm can take, while the average case time complexity is the algorithm's expected time. Sometimes an algorithm that has bad worst-case time complexity has good average-case time complexity, which means that it performs well on average, despite the potential for poor performance in specific scenarios.

To understand how asymptotic analysis helps evaluate the effectiveness of algorithms, let's dig deeper into the concept and how it applies to the examples of insertion sort and merge sort:

Efficiency and Scalability: Asymptotic analysis helps in understanding how the time or space requirements of an algorithm grow with the input size. This is critical to assess the efficiency of the algorithm and to understand how well it works. For example, Insertion Class has a time complexity of O(n^2). This means that if the size of the input is doubled, the time taken is likely to quadruple, which shows that it scales poorly with larger inputs. Merge sort, on the other hand, has a time complexity of O (n log n). If the size of the input is doubled, the time taken will slightly more than double, indicating that it scales much better for larger inputs.

Comparative analysis: Comparative analysis allows us to compare the performance of different algorithms and choose the most efficient one. For example, when comparing insertion order and merge sort, we can infer from the time complexities (O(n^2) vs. O (n log n)) that Merge Sort will be more efficient for large input sizes. However, for smaller inputs or roughly seeded inputs, insertion classification can be more efficient due to the lower overhead.

Understand trade-offs: Different algorithms may excel in different aspects and have different trade-offs. For example, while Merge Sort works better for larger inputs than Insertion Sort, it requires more space because it creates new arrays during the merge process. This may be a defect in environments with limited memory. On the other hand, Insertion Sort is a positional sorting algorithm that requires a fixed amount of extra space.

Set expectations: Knowing the time complexity of the algorithm, it sets an expectation of how long the process will take. If we know that the algorithm has a time complexity of O(n^2), we can expect it to be impractical for very large input sizes.

Worst, Average, and Best-case scenarios: The asymptotic analysis also considers the best, worst, and average cases of the algorithm. For example, an insertion sort has a best-case time complexity of O(n) when the input is already or is about to be sorted, while the worst-case is O(n^2) when the input is reverse-sorted. Big O notation provides an upper bound on an algorithm's performance, even in the worst-case scenario. For example, Insertion Sort has a worst-case time complexity of O(n^2), indicating that it may take significantly longer to sort a large, reverse-ordered dataset compared to Merge Sort's worst-case time complexity of O (n log n).

Standardized Metric: Big O notation provides a standardized metric for measuring algorithmic efficiency. For example, we can compare the time complexities of Insertion Sort (O(n^2)) and Merge Sort (O (n log n)) to objectively understand their relative efficiencies.

Algorithm Selection: Asymptotic analysis helps in algorithm selection for specific tasks. For instance, if we need to sort a large dataset, we would prefer Merge Sort with its time complexity of O (n log n) over Insertion Sort with its time complexity of O(n^2) to achieve faster results.

Optimization Strategies: Asymptotic analysis helps in identifying optimization strategies for improving algorithmic performance. In the case of Insertion Sort, which has a quadratic time complexity, we can explore optimizations like early termination or using binary search to

improve the insertion step. For Merge Sort, which already has an efficient time complexity, further optimizations may not be necessary.

Trade-Offs and Constraints: Asymptotic analysis helps understand trade-offs and constraints associated with different algorithms. For example, Insertion Sort has a better best-case time complexity of O(n), making it a favourable choice for small or nearly sorted data. Merge Sort, on the other hand, may require more memory due to its space complexity of O(n), which is a trade-off to achieve its efficient time complexity.

Algorithmic Complexity Classes: Asymptotic analysis and Big O notation play a crucial role in defining complexity classes. For instance, both Insertion Sort and Merge Sort fall under the comparison-based sorting algorithms, with Insertion Sort belonging to O(n^2) complexity class and Merge Sort belonging to O (n log n) complexity class. Understanding these complexity classes helps classify problems and guide the development of algorithms with known efficiency guarantees.

By incorporating Insertion Sort and Merge Sort as examples, we can see how asymptotic analysis and Big O notation are applied to evaluate and compare the efficiency of algorithms, aiding in algorithm selection, optimization, worst-case analysis, understanding trade-offs, and classifying problems based on their complexity.

**3.2** You have measured both sorting algorithms efficiency using time benchmarking technique. **Determine** other two ways to measure the algorithm efficiency, support your answer with examples.

Answer:

**1- Asymptotic analysis**

Asymptotic complexity analysis, also known as asymptotic analysis, is a method of describing the efficiency of algorithms when the input size grows towards infinity. It provides a method for analyzing the behavior of algorithms in terms of the required time and space. The term “asymptotic” means to approach a value or curve arbitrarily (i.e., when the input size becomes indefinitely large). This form of analysis is used because it allows a high-level simplified understanding of the algorithm's efficiency, regardless of machine-specific constants such as CPU speed.

Asymptotic analysis focuses on three main aspects:

Worst case complexity: the maximum time taken for any entry of size 'n'.

Average state complexity: The average time taken for all possible inputs of size 'n'.

Best case complexity: minimum time taken for any entry of size 'n'.

Big O notation is used to describe the upper bound or worst-case time complexity of an algorithm. This notation gives an upper limit on the time or space used by an algorithm in terms of input size 'n', as 'n' approaches infinity. In other words, it offers the worst possible algorithm performance. This is particularly useful because it can provide guarantees that the algorithm will not exceed a certain time or space limit.

For clarity, I will give some examples:

O (1): Constant time complexity. For example, getting the size of an array. No matter how big my array is, the process will take the same amount of time.

O(n): linear time complexity. A simple example is to search for a specific element in an array. In the worst-case scenario, I would need to look at each element once, so the time taken grows linearly with the size of the array.

O(n^2): quadratic time complexity. A common example is "bubble sort", where you need to compare each item with every other item in the list.

O (log n): Logarithmic time complexity. Merge Sort is a divide-and-conquer sorting algorithm. It works by dividing the unsorted list into n sublists, each containing one element (a list of one element is considered sorted), and then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining. This results in a sorted list.

In each of these cases, Big O notation provides an upper bound on the time complexity as the input size grows. This allows developers to understand how to extend the algorithm and helps decide which algorithm is most appropriate for a given context. It's important to note that Big O notation describes an upper bound, which means the actual runtime is guaranteed not to be worse than that, but possibly better.

**Merge Sort**

Merge Sort is a divide-and-conquer algorithm that splits the array into two halves, sorts them separately, and then merges them.

Time Complexity

Worst-case: O (n log n) - Even in the worst-case scenario, Merge Sort makes log(n) divisions of the array, and for each division, it takes O(n) time to merge the arrays. Hence, the worst-case time complexity is O (n log n).

Best-case: O (n log n) - Merge sort always divides the array into two halves and takes linear time to merge two halves.

Average-case: O (n log n) - Regardless of the initial ordering of the input, Merge Sort will have to perform the same steps.

Space Complexity

Merge Sort requires extra space to store the two halves during the merge process, which leads to a space complexity of O(n).

Example: If we have an array of 8 elements, Merge Sort would take approximately 24 operations in the worst-case scenario, and it would require space for 8 extra elements for the merge process.

**Insertion Sort.**

Time Complexity

Worst-case: O(n^2) - The worst case occurs if the input array is sorted in reverse order. Because each insertion is O(n), and we perform 'n' insertions, the worst-case time complexity is O(n^2).

Best-case: O(n) - The best-case occurs if the input array is already sorted. In this case, no reshuffle is needed in the inner loop, and the best-case time complexity is O(n).

Average-case: O(n^2) - On average, half of the elements in the array need to be reshuffled for each insertion.

Space Complexity

Since it's an in-place sorting algorithm, Insertion Sort doesn't require additional storage, leading to a space complexity of O (1).

Example: If we have an array of 8 elements, Insertion Sort could take up to 64 operations in the worst-case scenario (i.e., the array is sorted in reverse order), and it would only require space for one extra variable for swapping elements.

**2- Theoretical analysis** of data structures and algorithms are a critical aspect of computer science that provides an understanding of the performance and efficiency of algorithms based on mathematical models. Theoretical analysis focuses on deriving formulas to predict an algorithm's behaviour, not on actual implementation. This allows developers to reason about the scalability and efficiency of an algorithm based on the size of the input, without being bound by specific hardware or software factors.

This theoretical analysis usually revolves around two primary metrics - time complexity and space complexity.

Time Complexity: This measures the amount of time an algorithm takes to run as a function of the size of the input. It is typically expressed using Big O notation, which describes the upper bound of the time complexity in the worst-case scenario.

Space Complexity: This measures the total amount of memory space that the algorithm needs relative to the size of the input. It includes both the space needed by the input data and the additional space used by the algorithm (like auxiliary variables, stack space, etc.).

Now, let's examine how theoretical analysis helps in assessing the efficiency of algorithms using the examples of Insertion Sort and Merge Sort:

Insertion Sort: This sorting algorithm works by building a sorted array or list one item at a time. It is much less efficient on large lists than more advanced algorithms like QuickSort, HeapSort, or MergeSort. The worst-case and average time complexity of Insertion Sort is both O(n^2), where n is the number of items being sorted. This happens when the input array is in reverse order. For a list that is already sorted, the time complexity is O(n). The space complexity is O (1) because only a single additional memory space is required.

Merge Sort: Merge Sort is an efficient, stable sorting algorithm that makes use of the divide-and-conquer principle. It works by dividing the unsorted list into n sublists, each containing one element, then repeatedly merging sublists to produce new sorted sublists until there is only one sublist remaining. This algorithm is more efficient than Insertion Sort for larger lists. The worst-case and average time complexities of Merge Sort are both O (n log n), regardless of the order of the input array. Its worst-case space complexity is O(n) due to the auxiliary space used during the merge process.

Theoretical analysis allows us to predict that Merge Sort will outperform Insertion Sort for larger input data sizes, even before implementing them. The time complexity of Insertion Sort is quadratic, while Merge Sort has a linear-logarithmic time complexity. Consequently, as the size of the input data grows, the time taken by Insertion Sort will increase much more rapidly than the time taken by Merge Sort.

However, when dealing with smaller datasets or nearly sorted datasets, Insertion Sort could outperform Merge Sort. The constant factors and lower order terms that Big O notation ignores can play a significant role for smaller or nearly sorted input data. In addition, Insertion Sort has a lower space complexity compared to Merge Sort, making it a more memory-efficient algorithm.

It's important to note that theoretical analysis gives a high-level understanding of the algorithm's behaviour and does not take into account the specifics of a machine or compiler, which might affect the actual performance. Therefore, it's essential to pair theoretical analysis with experimental analysis to get a comprehensive understanding of an algorithm's performance.

In conclusion, theoretical analysis is an essential tool that gives us the ability to predict an algorithm's performance concerning time and space based on the size of the input data. It helps developers choose the most efficient algorithm according to the problem's constraints, ensuring effective and efficient solutions.

**3.3** **Discuss** the performance both behaviours of algorithms in relation to their implementation, complexity, array sizes, different array shapes, and evaluate the obtained outcomes against the intended ones.

Answer:

First:

Creating the array to be sorted:

It first defines an array of a specific size (N = 100,000). It calculates and prints the time it takes to sort each of these arrays. These steps have time complexity O(N) because we're simply iterating over the array once, and space complexity is also O(N) as we're storing N values in the array.

Defining the mergeSort function:

This is a recursive function that takes as input an array and divides it into two halves until it gets to arrays of length 1 (which are by definition sorted). It then merges these arrays in sorted order.

Time Complexity: The time complexity of merge sort is O(NlogN) in all cases. This is because the array is being split logN times (this is the number of times you can halve N before you get to 1), and for each split, we do a linear amount (N) of work to merge the arrays.

Space Complexity: The space complexity is also O(NlogN). In each recursive call, we create left and right arrays, hence we need additional space. However, these additional spaces are freed up after each merge operation completes. So, the maximum space used at any point would be proportional to N (for storing the array) plus the depth of the recursive call stack, which is logN. Hence, it's O (N + logN), but we drop the logN term as N dominates for large inputs, so it's O(N).

Defining the merge function:

This function takes as input the original array and the two halves into which it has been divided. It then sorts the original array by comparing the values of the two halves one by one, starting from the beginning. If one half is exhausted before the other, it simply dumps all the remaining values from the other half into the original array.

Time Complexity: The time complexity of the merge operation is O(N), where N is the total number of elements in the left and right arrays. The function does a constant amount of work (comparing elements and assigning them to the array) for each element in the two arrays.

Space Complexity: The space complexity of the merge function is O (1), or constant space, if we disregard the space needed to hold the result (as the result needs to be stored anyway). This is because it only uses a few integer variables and does not depend on the size of the input arrays. However, if we include the space for the output array, the space complexity would be O(N) for this function, where N is the size of the 'array’.

This is an implementation of the Insertion Sort algorithm. The basic idea behind Insertion Sort is to divide the array into a sorted and an unsorted region. The sorted region starts with the first element and the unsorted region with everything to the right of it. We gradually expand the sorted region to the right by inserting the first element from the unsorted region at the correct place in the sorted region.

Here's the detailed breakdown of the steps:

The outer for loop runs from the second element (index 1) to the last element. For each iteration of this loop, we consider the current element (a[i]) as the key.

We compare this key with the elements before it (a[j]). If a previous element is greater than the key, it means the key should be located somewhere before this previous element. So, we shift the previous element one position to the right (a[j+1] = a[j]) and decrease j to continue our search for the correct spot.

If the previous element is not greater than the key (or if we've reached the start of the array), we've found the correct spot for the key. We place it there (a[j+1] = key).

Time Complexity:

Worst-case scenario is O(n^2). This is when the input array is reverse sorted. We have to shift elements of the sorted region for each of the n elements in the unsorted region.

Best-case scenario is O(n). This is when the input array is already sorted. In this case, we don't have to shift any elements (inner while loop is not executed). Average-case complexity is also O(n^2).

Space Complexity:

The space complexity of the Insertion Sort algorithm is O (1), which means it requires a constant amount of additional space. This is because it only uses a single additional memory space to temporarily hold the variable 'key', which does not change with the size of the input array. Hence, it is considered an "in-place" sorting algorithm.

Merge Sort outperforms Insertion Sort on larger, sorted datasets due to their inherent differences in time complexity. This is evident in the data provided for a sorted array with 100,000 elements where Merge Sort completed in 18,037,700 nanoseconds and Insertion Sort completed in 1,884,323,100 nanoseconds.

The time complexity for Insertion Sort is O(n^2), which is quadratic. It performs well on small datasets or datasets that are nearly sorted, because it sorts in place and requires a single pass through the data. However, its performance degrades quickly as the size of the dataset increases. In the worst-case scenario (when the array is reverse sorted), each insertion is at the maximum position it can be, which is why you see such a large runtime for larger, sorted datasets.

On the other hand, Merge Sort has a time complexity of O (n log n), which is logarithmic. This means the number of operations grows more slowly than the dataset size. Merge Sort divides the dataset into smaller parts, sorts them, and then merges them together. This divide and conquer strategy allow Merge Sort to handle larger datasets more effectively.

The concept of time complexity is a measure of the amount of computational work required to complete a task. It is a function of the size of the input, denoted as n. Lower time complexity generally indicates better performance on larger datasets, which is why Merge Sort is faster than Insertion Sort for larger, sorted datasets.

The specific nature of Insertion Sort (with its best-case time complexity of O(n) on an already sorted array) is such that it tends to perform poorly as the size of a sorted dataset increases, because it must still traverse the entire dataset. Merge Sort, in contrast, will continue to divide the problem, reducing the amount of work done at each step, resulting in much faster sorting times for larger datasets.

Merge Sort's divide-and-conquer approach and resultant O (n log n) time complexity enable it to handle large datasets more efficiently than Insertion Sort. It's better suited for large, sorted datasets, where the number of comparisons and moves becomes prohibitively expensive with algorithms that have quadratic time complexity, like Insertion Sort.

The performance of a sorting algorithm is heavily influenced by its time complexity. Time complexity is a computational complexity that describes the amount of computer time taken by an algorithm to run, as a function of the size of the input to the program. The time complexity of algorithms is most commonly expressed using Big O notation, which describes the upper bound of the time complexity in the worst-case scenario.

Merge Sort has a time complexity of O (n log n), both in average and worst-case scenarios. As an algorithm, Merge Sort follows a divide-and-conquer strategy. It repeatedly divides the array into halves, sorts them separately, and merges them. This approach gives Merge Sort the advantage when dealing with large datasets.

On the other hand, Insertion Sort has a time complexity of O(n^2) in the worst-case scenario (when the input array is reverse sorted), and in average-case scenarios. For already sorted arrays (best case), Insertion Sort has a linear time complexity, O(n). However, the best-case scenario is rarely assumed because it’s not a practical situation. In reality, data is usually not sorted.

In terms of the order of growth, O(n^2) (Insertion Sort) grows faster than O (n log n) (Merge Sort) as n increases. This means that as the input size increases, the time taken by Insertion Sort increases much faster than Merge Sort.

For example, let's consider sorting an array with 100,000 elements. For Merge Sort, the order of growth is O (n log n), which translates into 100,000 \* log2(100,000) ≈ 1,665,897 operations. For Insertion Sort, the order of growth is O(n^2), which is 100,000^2 = 10,000,000,000 operations in the worst case. This difference is significant and shows why Merge Sort outperforms Insertion Sort on larger datasets.

It's important to note that these are theoretical analyses. Many other factors can influence performance in practice, including: implementation details, memory hierarchy effects, and the architecture of the specific machine where the code is running.

In conclusion, due to its higher time complexity, Insertion Sort performs significantly worse than Merge Sort for larger datasets. While Insertion Sort can be efficient for smaller or nearly sorted datasets, Merge Sort is a better general-purpose sorter due to its consistent O (n log n) performance across different input arrangements and its superior scalability for large inputs.

Insert Sorting is preferred over Merge Sort for reverse sorted arrays because it works faster due to the nature of its implementation and its time complexity in this specific case.

The time complexity of insertion sort is usually O(n^2) in its general case. This means that the number of operations it performs grows quadratic with the size of the input. However, the implementation I made of Insertion Sort is designed to work in descending order. So, when given a reverse sorted array (sorted in descending order), Insertion Sort behaves as if it were dealing with an already sorted array, causing it to act in linear time, O(n). This is because it only needs to do one comparison per item, realizing that each item is already in place. Thus, he can quickly traverse the matrix.

On the other hand, Merge Sort, which usually has a time complexity of O (n log n), doesn't make much use of an input array that's already sorted, either in ascending or descending order. Even if the input is sorted, the Merge Sort still needs to split the array into one-element arrays and then merge them back together, which requires log(n) partitions and n log(n) comparisons and swaps. As a result, it works slower than Insertion Sort defined on a reverse sorted array.

However, it is important to note that while Insertion Sort may perform better on this particular entry, Merge Sort will generally be a better option for larger datasets or those of no particular order, due to their generally better time complexity.

In terms of the order of growth, O(n) (Insertion Sort in this case) is better than O (n log n) (Merge Sort), which means that for larger inputs, the time taken by an algorithm with linear time complexity will grow slower of those with linear logarithmic time complexity. Thus, in a large, inversely sorted array, your specific implementation of Insertion Sort outperforms Merge Sort.

In the case of randomly sorted arrays of size N = 100,000 Merge Sort is significantly superior to Insertion Sort.

Merge sort is a divide-and-conquer algorithm that divides the input array into smaller and smaller parts, sorts these pieces, and then merges them together in the correct order. Its performance is very consistent, with a time complexity of O (n log n) under all scenarios, including the worst case. This time complexity is derived from the fact that the algorithm continuously splits the array into two parts (which gives the log N part) and then has to look at each element once when it merges everything together (which gives the n part).

In contrast, Insertion Sort is a simple comparison-based algorithm that, for each element in the array, recursively swaps it with the previous element until it is in the correct position. In the best-case scenario (array already sorted), this option gives Insertion Sort a time complexity of O(n). However, in the worst-case scenario (reverse sorted array) and medium case scenario (randomly sorted array), the time complexity of insertion sort degenerates to O(n^2). This is because, for each item, it will likely have to compare and alternate with every item that came before it.

Since the time complexity of insertion sort is quadratic proportional to the input size, and the time complexity of merge sort is linearly proportional to the input size, it is not surprising that merge sort outperforms classification on larger randomly ordered arrays. For small arrays or roughly sorted arrays, the difference will be less noticeable, and insertion order may be faster due to the lower overhead. But as the array size increases, the efficiency of merge sort becomes more and more obvious.

So, in the case where you have an array size of 100,000, the efficiency of merge sort makes it a much better choice for randomly sorted data. That's why I'm seeing it perform much faster than sorting insertions into the array of the random classifier.

Therefore, in terms of the order of growth, Insertion Sort's time complexity (O(n^2)) grows much faster than Merge Sort's time complexity (O (n log n)) as the size of the input increases. This means that, for large inputs, the time taken by Insertion Sort will grow at a much faster rate than that of Merge Sort.

Merge Sort is generally a better choice for large, randomly sorted arrays. While Insertion Sort might have less overhead and thus be faster for smaller or nearly sorted arrays, its time complexity makes it less suited for large, randomly sorted arrays.

**In mergeSort I compared and explained three different shapes of arrays:**

Sorted Array: 18037700 nanoseconds: Given that the array is initially sorted in ascending order, this is the worst-case scenario for this particular Merge Sort implementation because it's designed to sort in descending order. Therefore, the algorithm has to perform a significant amount of work to reverse the order of the elements, which is why it takes the most time in this case.

Reversely Sorted Array: 16136900 nanoseconds: In contrast to the sorted array, the reversely sorted array is essentially already sorted in the desired descending order. As such, the Merge Sort algorithm doesn't have to do as much work, resulting in a quicker sort and the smallest time in nanoseconds. This behavior differs from a typical Merge Sort implementation where the initial state of the data (sorted, reverse sorted, random) does not affect the time complexity.

Random Array: 21520300 nanoseconds: For a random array, the Merge Sort has to organize the data from an unordered state to a descending order. It's a more demanding task than dealing with a reverse-sorted array, but less so than with an ascending sorted array, which explains why the time taken falls between the other two scenarios.

In summary, this particular Merge Sort implementation works best with data that's already sorted in descending order (reverse sorted). It's a clear demonstration of how the initial state of data and the specific characteristics of an algorithm can influence its performance and efficiency. In terms of time complexity and growth order, Merge Sort consistently works at a level of O (n log n), making it efficient for large datasets. However, in this particular case, the nature of the data can also impact the performance due to the sorting order preference in the algorithm design.

**In Insertion sort I compared and explained three different shapes of arrays:**

Sorted Array: 1884323100 nanoseconds: The sorted array is in ascending order, which is the worst-case scenario for this particular Insertion Sort implementation. This happens because the algorithm is sorting in the opposite direction (descending), so it has to move a lot of elements around to get the sorted array in the desired order. Hence, it has to do the maximum amount of work, leading to the worst time complexity of O(n^2).

Reversely Sorted Array: 1409200 nanoseconds: The reverse sorted array is essentially already sorted in the desired order (descending), so the Insertion Sort doesn't have much work to do, and can simply iterate through the data to confirm it is sorted. Thus, it behaves in nearly linear time complexity, O(n), in this case, which is the best-case scenario for Insertion Sort and results in the lowest number of nanoseconds.

Random Array: 773637700 nanoseconds: A random array is neither sorted nor reversely sorted, so the time complexity for Insertion Sort is expected to be O(n^2), as the algorithm will have to perform a large number of comparisons and swaps. However, it's less time-consuming than sorting an ascending ordered array because the data might partially be in the desired descending order.

In summary, this particular Insertion Sort implementation works best with data that's already sorted in descending order (reverse sorted). Despite having a quadratic time complexity of O(n^2) in its average and worst cases, it can perform well in nearly linear time, O(n), when the input is already sorted in the same order it's designed to sort. This reflects the adaptability of Insertion Sort but also underscores its inefficiency with larger datasets or those that are far from the sorted order.

Task 4:

4.1: Define what is the closest MacDonalds branch for the delivery man to pick the order from for each client. Show detailed tracing using *bellman ford* and *Dijkstra's* algorithm shortest path algorithms.

Answer:

Bellman ford algorithm

Branch 1:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Edge** |
| **B1** | 0 | ------- |
| **B2** | ~~Infinity~~ 3 | B1🡪 B2 |
| **B3** | ~~Infinity~~ 1 | B1🡪 B3 |
| **B4** | ~~Infinity~~ 10 | B1🡪 B4 |
| **B5** | ~~Infinity 5~~ 4 | ~~B2B5~~ B3🡪 B5 |
| **C1** | ~~Infinity~~ 2 | B3🡪 C1 |
| **C2** | 0 | B1🡪 C2 |

[B1B2, B1B3, B1B4, B1C2, B2B5, B3B5, B3C1, B4B3, B4C1, B5C2, C1C2]

Because there are 7 vertices in branch 1, in the worst case we have 6 iterations, but only because after second iterations all the vertices were relaxed and distances remained the same, here we stopped after the second iteration.

Branch 2:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Edge** |
| **B2** | 0 | ------- |
| **B1** | Infinity |  |
| **B3** | Infinity |  |
| **B4** | Infinity |  |
| **B5** | ~~Infinity~~ 2 | B2 🡪 B5 |
| **C1** | Infinity |  |
| **C2** | ~~Infinity~~ 7 | B5 🡪 C2 |

**[B2B5, B1C2, B1B3, B1B4, B1B2, B3C1, B3B5, B4C1, B4B3, B5C2, C1C2]**

Because there are 7 vertices in branch 2, in the worst case we have 6 iterations, but only because after second iterations all the vertices were relaxed and distances remained the same, here we stopped after the second iteration.

Branch 3:

|  |  |  |
| --- | --- | --- |
| Vertex | Distance | Edge |
| B3 | 0 | ------- |
| B1 | Infinity |  |
| B2 | Infinity |  |
| B4 | Infinity |  |
| B5 | ~~Infinity~~ 3 | B3🡪 B5 |
| C1 | ~~Infinity~~ 1 | B3🡪 C1 |
| C2 | ~~Infinity 8~~ 3 | ~~B5C2~~ C1🡪 C2 |

**[B3C1, B3B5, B1B2, B1C2, B1B3, B1B4, B2B5, B4B3, B4C1, B5C2, C1C2]**

Because there are 7 vertices in branch 3, in the worst case we have 6 iterations, but only because after second iterations all the vertices were relaxed and distances remained the same, here we stopped after the second iteration.

Branch 4:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Edge** |
| **B4** | 0 | ------- |
| **B1** | Infinity |  |
| **B2** | Infinity |  |
| **B3** | ~~Infinity~~ 1 | B4🡪 B3 |
| **B5** | ~~Infinity~~ 4 | B3🡪B5 |
| **C1** | ~~Infinity 4~~ 2 | ~~B4C1~~ B3🡪 C1 |
| **C2** | ~~Infinity 9~~  ~~4~~ | ~~B5C2~~  C1🡪 C2 |

**[B4B3, B4C1, B1B2, B1C2, B1B3, B1B4, B2B5, B3B5, B3C1, B5C2, C1C2]**

Because there are 7 vertices in branch 4, in the worst case we have 6 iterations, but only because after second iterations all the vertices were relaxed and distances remained the same, here we stopped after the second iteration.

Branch 5:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Edge** |
| **B5** | 0 | ------- |
| **B1** | Infinity |  |
| **B2** | Infinity |  |
| **B3** | Infinity |  |
| **B4** | Infinity |  |
| **C1** | Infinity |  |
| **C2** | ~~Infinity~~ 5 | B5 🡪 C2 |

**[B5C2, B1B2, B1C2, B1B3, B1B4, B2B5, B3B5, B3C1, B4B3, B4C1, C1C2]**

Because there are 7 vertices in branch 5, in the worst case we have 6 iterations, but only because after second iterations all the vertices were relaxed and distances remained the same, here we stopped after the second iteration.

After considering all the branches in the scenario, I have concluded that **branch 3** is the closest one to **C1**, and the **branch 1** is the closest one to **C2** using **Bellman ford** algorithm.

## Bellman Ford Pseudocode:

* **Initialization:**
  + The algorithm starts with a graph G and a source vertex S.
  + For each vertex V in G, it initializes the distance to that vertex as infinity and sets the predecessor vertex of V as NULL.
  + It sets the distance to the source vertex S as 0, since the distance from a vertex to itself is always zero.
* **Relaxation:**
  + For each vertex in G, it performs the relaxation process for all the edges in G. The relaxation process checks all the edges (U, V) and if the distance to V can be shortened by going through U, it updates the shortest distance distance[V] and sets U as its predecessor.
* **Negative Cycle Detection:**
  + After the relaxation process for all vertices, it again checks all the edges (U, V). If it can still find an edge that can update distance[V], it means there is a negative cycle in the graph (a cycle where the sum of the weights of the edges is negative). The algorithm throws an error in this case, because it cannot find the shortest path in a graph with a negative cycle.
* **Output:**
  + If no negative cycles are detected, the algorithm returns two arrays: distance[], which contains the shortest path distances to each vertex, and previous[], which contains the predecessor of each vertex on the shortest path.

function bellmanFord(G, S)

for each vertex V in G

distance[V] <- infinite

previous[V] <- NULL

distance[S] <- 0

for each vertex V in G

for each edge (U,V) in G

tempDistance <- distance[U] + edge\_weight(U, V)

if tempDistance < distance[V]

distance[V] <- tempDistance

previous[V] <- U

for each edge (U,V) in G

If distance[U] + edge\_weight(U, V) < distance[V}

Error: Negative Cycle Exists

return distance[], previous[]

* G: Represents the Graph. This is the set of vertices (nodes) and their connections (edges) that the algorithm is working with.
* V: Represents a Vertex (singular) in the graph G. Vertices are essentially the nodes or points in the graph.
* S: Stands for the Source vertex. This is the starting point from which the shortest paths to all other vertices in the graph are calculated.
* U: Represents a different vertex in the graph. In the context of the edge (U, V), U is the vertex from which an edge is coming, and V is the vertex where the edge is going. The algorithm checks whether it's more efficient to reach V by passing through U compared to the currently known shortest path to V.

Dijkstra's algorithm

Branch 1:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Parent** |
| **B1** | 0 | ------- |
| **B2** | ~~Infinity~~ 3 | B1 |
| **B3** | ~~Infinity~~ 1 | B1 |
| **B4** | ~~Infinity~~ 10 | B1 |
| **B5** | ~~Infinity~~ 4 | B3 |
| **C1** | ~~Infinity~~ 2 | B3 |
| **C2** | ~~Infinity~~ 0 | B1 |

|  |  |
| --- | --- |
| **Priority Queue** | |
| **~~B1~~** | **~~B2~~** | **~~B3~~** | **~~B4~~** | **~~C2~~** | **~~B5~~** | **~~C1~~** |

|  |  |
| --- | --- |
| **Visited** | |
| **B1** | **C2** | **B3** | **C1** | **B2** | **B5** | **B4** |

Branch 2:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Parent** |
| **B2** | 0 | ------- |
| **B1** | Infinity |  |
| **B3** | Infinity |  |
| **B4** | Infinity |  |
| **B5** | ~~Infinity~~  2 | B2 |
| **C1** | Infinity |  |
| **C2** | ~~Infinity~~  7 | B5 |

|  |  |
| --- | --- |
| **Priority Queue** | |
| **~~B2~~** | **~~B5~~** | **~~C2~~** |

|  |  |
| --- | --- |
| **Visited** | |
| **B2** | **B5** | **C2** |

Branch 3:

|  |  |  |
| --- | --- | --- |
| Vertex | Distance | **Parent** |
| B3 | 0 | ------- |
| B1 | Infinity |  |
| B2 | Infinity |  |
| B4 | Infinity |  |
| B5 | ~~Infinity~~ 3 | B3 |
| C1 | ~~Infinity~~ 1 | B3 |
| C2 | ~~Infinity~~ 3 | C1 |

|  |  |
| --- | --- |
| **Priority Queue** | |
| **~~B3~~** | **~~B5~~** | **~~C1~~** | **~~C2~~** |  |  |  |

|  |  |
| --- | --- |
| **Visited** | |
| **B3** | **C1** | **B5** | **C2** |  |  |  |

Branch 4:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Parent** |
| **B4** | 0 | ------- |
| **B1** | Infinity |  |
| **B2** | Infinity |  |
| **B3** | ~~Infinity~~ 1 | B4 |
| **B5** | ~~Infinity~~ ~~4~~ | B3 |
| **C1** | ~~Infinity 4~~  2 | B4 |
| **C2** | ~~Infinity~~  4 | C1 |

|  |  |
| --- | --- |
| **Priority Queue** | |
| **~~B4~~** | **~~B3~~** | **~~C1~~** | **~~B5~~** | **~~C2~~** |  |  |

|  |  |
| --- | --- |
| **Visited** | |
| **B4** | **B3** | **C1** | **B5** | **C2** |  |  |

Branch 5:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Distance** | **Parent** |
| **B5** | 0 | ------- |
| **B1** | Infinity |  |
| **B2** | Infinity |  |
| **B3** | Infinity |  |
| **B4** | Infinity |  |
| **C1** | Infinity |  |
| **C2** | ~~Infinity~~ 5 | B5 |

|  |  |
| --- | --- |
| **Priority Queue** | |
| **~~B5~~** | **~~C2~~** |

|  |  |
| --- | --- |
| **Visited** | |
| **B5** | **C2** |

After considering all the branches in the scenario, I have concluded that **branch 3** is the closest one to **C1**, and the **branch 1** is the closest one to **C2** using ***Dijkstra's*** algorithm.

**Dijkstra's Pseudocode:**

* Initialization:

The algorithm starts with a graph G and a source vertex S.

For each vertex V in G, it initializes the distance to that vertex as infinity, and the predecessor vertex of V as NULL.

It adds all vertices to a Priority Queue Q, except for the source S. The distance for S is set to 0.

* Iteration:

As long as Q is not empty, it extracts the vertex U with the minimum distance from Q.

Then, for each unvisited neighbor V of U, it calculates a temporary distance, which is the sum of the distance to U and the edge weight between U and V.

If this temporary distance is less than the current distance to V, it updates the distance to V and sets U as the predecessor of V.

* Output:

Once the Priority Queue is empty (i.e., all vertices have been visited), the algorithm ends. It returns two arrays: distance[] which stores the shortest path distances to each vertex, and previous[] which stores the predecessor of each vertex on the shortest path.

This algorithm is widely used in network routing protocols, among other applications. It essentially maintains an array to store the shortest path from the source vertex to each vertex in the graph, and a priority queue to select the vertex with the shortest path for expansion.

At the end, you can find the shortest path to a destination vertex by backtracking from the destination to the source using the previous[] array.

**G:** Stands for Graph. It represents the set of vertices (nodes) and their connections (edges) that the algorithm is working with.

**V:** Represents a Vertex (singular) in the graph G. Vertices are the nodes or points in the graph.

**S:** Represents the Source vertex. This is the starting point from which the shortest paths to all other vertices in the graph are calculated.

**Q:** Represents a Priority Queue. This is a type of queue where elements (in this case, vertices) are removed based on their priority (in this case, the current shortest known distance from the source vertex S). In Dijkstra's algorithm, it's used to efficiently select the next vertex to visit.

function dijkstra(G, S)

for each vertex V in G

distance[V] <- infinite

previous[V] <- NULL

If V != S, add V to Priority Queue Q

distance[S] <- 0

while Q IS NOT EMPTY

U <- Extract MIN from Q

for each unvisited neighbour V of U

tempDistance <- distance[U] + edge\_weight(U, V)

if tempDistance < distance[V]

distance[V] <- tempDistance

previous[V] <- U

return distance[], previous[]

4.2: Critically analyse the complexity of both algorithms in reference to their DS implementation.

Answer:

**Dijkstra's Algorithm:**

First loop: "Dequeue" operation: For every node in the graph, we need to find the node with the smallest tentative distance that has not been visited yet. If we are using an array or an adjacency matrix for this, we would have to scan through all nodes (V operations) to find this minimum distance node. Since we need to find this node for all vertices in the graph, we perform this operation V times, hence it becomes a O(V) operation.

Second loop: "Update Distances" operation: For the current node, we update all its neighbours. In the worst case, this can be all other nodes in the graph (V operations). This process is also performed for all nodes, leading to another O(V) operation.

So, the overall time complexity, when using arrays or adjacency matrices, becomes O(V) \* O(V) = O(V^2).

However, it's important to remember that this is not the most efficient implementation of Dijkstra's algorithm. If we use a more efficient data structure like a binary heap in a priority queue, we can improve the time complexity to O((V+E) log V), where E is the number of edges in the graph. This is because extracting the minimum distance node and updating distances become logarithmic operations.

For a sparse graph (where E is much less than V^2), this improvement can have a substantial impact on the performance of the algorithm. If the graph is dense (E is close to V^2), the O(V^2) and O((V+E) log V) time complexities become comparable.

**Bellman-Ford Algorithm:**

The Bellman-Ford algorithm uses a single array to keep track of the shortest distance from the source node to every other node in the graph. The algorithm iterates over all edges in the graph, and for each edge, it checks whether the edge provides a shorter path to the destination node than the current shortest known path. This process is repeated V-1 times, where V is the number of vertices in the graph, ensuring that each shortest path has been finalized.

The time complexity for Bellman-Ford is O(V\*E), where V is the number of vertices and E is the number of edges. This is because, for every vertex, we relax all edges, which results in this time complexity. Thus, even though we are using a simple data structure (array), the algorithm can still be relatively slow, especially for dense graphs.

References:

Munish Chandel (2018). *What are four basic principles of Object Oriented Programming?* [online] Medium. Available at: https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727.

GeeksforGeeks. (2022). *Applications, Advantages and Disadvantages of Queue*. [online] Available at: https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-queue/?ref=gcse.

GeeksforGeeks. (2011). *Applications of Queue Data Structure*. [online] Available at: https://www.geeksforgeeks.org/applications-of-queue-data-structure/?ref=gcse [Accessed 24 May 2023].

Tutorialspoint.com. (2019). Data Structures - Asymptotic Analysis - Tutorialspoint. [online] Available at: https://www.tutorialspoint.com/data\_structures\_algorithms/asymptotic\_analysis.htm.

Wright, G. (n.d.). What is empirical analysis? - Definition from WhatIs.com. [online] WhatIs.com. Available at: https://www.techtarget.com/whatis/definition/empirical-analysis.

www.programiz.com. (n.d.). *Bellman Ford’s Algorithm*. [online] Available at: <https://www.programiz.com/dsa/bellman-ford-algorithm>.

Programiz (2019). *Dijkstra’s Algorithm*. [online] Programiz.com. Available at: <https://www.programiz.com/dsa/dijkstra-algorithm>.

‌

‌

‌

**Plagiarism**

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

**Student Declaration**

|  |
| --- |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.  Student signature: Rashed Hasan Qahah Date: |