DSA Practice Problems

Quiz 2:
Chapter 4: 5,6,9,10,12,14 (pages 235-237) Chapter
5: 2,3,7,9,12,14,19 (pages 303-307)
Quiz 3 Chapter 7: 3,8,9,11,14 (pages 395-398)
Chapter 8: 3,5,6,7,8,13,14 (pages 446-448)
Quiz 4 Chapter 3: 4, 10, 11, 12, 13, 14, 15, 18, 23
(pg 189-193)
Quiz 5 Chapter 10: 1,5,6,7,11,12,13,14,15,18,20,23
(pg 554-557)

Chapter 4: 5,6,9,10,12,14 (pages 235-237)

Ex. 5:

Each operation can be performed on an ADT bag provided that the bag will keep track of two variables which are initialized at the creation of the bag, a count of how many items it has inside the bag and the maximum number of items that can be in the bag at the same time. The count variable is initialized to 0 and the max variable is initialized to a number provided in the ADT's constructor.

```
add(Item i)
```

The **add** method will add an object of the type **Item** into the ADT bag.
Precondition: The calling code will provide the method with a parameter of the type **Item**. The code also has to make sure that the bag is not completely full yet.
Postcondition: The method should return a Boolean value after executing, it should return **true** if the Item was added successfully and return **false** if it is not provided with a parameter of an Item type or if the bag is full. The method will also update the item counter inside the ADT bag by adding 1 if the operation was successful.

```
removeLast()
```

The removeLast method will remove the most recently added item to the bag or return **false** if the bag is empty.
Precondition: The calling code does not have to provide the method with any parameters.
Postcondition: This method should return a Boolean value after executing, it should return **true** if the operation was completed successfully and it should return **false** if the bag was empty or if it was called before the first **add** method call. This method will also update the item counter inside the ADT bag by subtracting 1 if the operation was successful.

```
removeRandom()
```

The **removeRandom** method will randomly choose an item in the bag and remove it from the collection or return **false** if the bag is empty.
Precondition: The calling code does not have to provide any arguments to this method.
Postcondition: This method should also return a Boolean value after executing. The method should return **true** if an Item was removed and return **false** if the bag was empty and there was no Item to remove. This method should also update the item counter by subtracting 1 if the operation was successful.

`countItems()`

The **countItems** method will return the number of items that are currently in the bag.
Precondition: The calling code does not have to provide any arguments to this method.
Postcondition: This method will return the item count variable of the bag. This count is initialized to 0 when the bag is created so it will simply return 0.

`isEmpty()`

The **isEmpty** method will return whether or not a bag is empty.
Precondition: The calling code does not have to provide any arguments to this method.
Postcondition: the method will return a Boolean stating whether the bag is empty. The method will return **true** when the **count** variable is 0 and will return **false** otherwise.

`removeAll()`

The **removeAll** method will remove all of the items inside the bag and will reset the **count** variable to 0.
Precondition: The bag must have at least one item in it.
Postcondition: The method will return a Boolean stating whether the operation was successful or not. It will return **false** if no items were in the bag at the time that the method was called.

Ex. 6:

• Define the CreditCard class ADT.

o Declare the data members to hold the credit card details as,

• String variable to hold name,

• long variable to hold account number,

• String variable to hold due date,

• int to hold the points of reward

• double to hold the balance of the credit card.

o Define the operations as,

• A constructor to initialize the data members to the default values.

• A set method to set the data members values with the client passed parameter values.

• Getter methods to retrieve the data of the data members.

• A charge method to add charges to the balance.

• An advanced method to add advance taken by the user.

• A payment method to deduct the amount paid by the user from the amount of credit card.

• An interest method to add interest of the credit card to the existing balance.

• A display method to display the values of the data members in a format to provide detailed information to the user.

• Design the Test class to test the operations of one credit card user.

```
Public class CreditCard
{
      // data members
      // to hold the account holder name
      Private String name;

      // to hold the account number
      Private long accountNumber;

      // to hold the due date of the account
      Private String dueDate;

      // to hold the reward points earned by the account
      Private int rewardPoints;


      // to hold the balance of the account
      Private double accountBalance;

      // constructor of the CreditCard which sets to the default
values
      Public CreditCard()
      {
            setValues(“ ”, 00000, “01/01/2018”. 0, 0.00);
      }

      // to initialize the data members with the parameter values
      Public void setValues(String name, long accountNumber, String
dueDate, int rewardsPoints, double accountBalance)
      {
            this.name = name;
            this.accountNumber = accountNumber;
            this.dueDate = dueDate;
            this.rewardPoints = rewardPoints;
            this.accountBalance = accountBalance;
      }

      // chargeCreditCard() method to add charges to the balance of
      the card
      Public void chargeCreditCard(double charge)
      {
            this.accountBalance = this.accountBalance + charge;
      }

      // to get the name of the credit card holder
      Public String getName()
      {
            return name;
      }

      // to get the account number
      Public long getAccountNumber()
      {
            return accountNumber;
      }

      // to get the reward points
      Public int getRewardPoints()
      {
            return rewardPoints;
      }
```

```
	// to get the account balance
	Public double getAccountBalance()
	{
		return accountBalance;
	}

	// submitPayment() method to pay the payment of the card
	Public void submitPayment(double payment)
	{
		this.accountBalance = this.accountBalance - payment;
	}

	// addInterest() method to add interest amount to card balance
	Public void addInterest(double interest)
	{
		this.accountBalance = this.accountBalance + interest;
	}

	// displayStatistics() to display the stats of the card
	Public void displayStatistics()
	{
		System.out.println("Account number : " + accountNumber);
		System.out.println("Account Holder Name: " + name);
		System.out.println("Due Date: " + dueDate);
		System.out.println("Reward Points: " + rewardPoints);
		System.out.println("Account Balance: $" + accountBalance);
	}
}



	DRIVER:
	Public class CreditCardImplementation
	{
		Public static void main(String args[])
		{
			CreditCard card = new CreditCard();
			card.setValues("Tony Rosado", 12541254, "04/18/2023",
	120, 2000);
			card.displayStatistics();
			card.chargeCreditCard(10);
			System.out.println("\n The out standing balance " +
			"after adding charges is: $" +
			card.getAccountBalance());
			card.submitPayment(500);
```

```
                    System.out.println("\n The out standing balance " +
                    "after making payment is: $" +
                    card.getAccountBalance());
                    card.addInterest(150);
                    System.out.println("\n The out standing balance " +
                    "after making payment is: $" +
                    card.getAccountBalance() + "\n");
                    card.displayStatistics();
            }
        }
        SAMPLE OUTPUT:
        Account Number: 12541254
        Account Holder Name: Tony Rosado
        Due Date: 04/18/2023
        Reward Points: 120
        Account Balance: $2000.0

        The out standing balance after adding charges is: $2010.0
        The out standing balance after making payment is: $1510.0
        The out standing balance after making payment is: $1660.0

        Account Number: 12541254
        Account Holder Name: Tony Rosado
        Due Date: 04/18/2023
        Reward Points: 120
        Account Balance: $1660.0
```

Ex. 9-10:

```
Class Trapezoid
{
        Double height;
        Double base1;
        Double base2;
        Double leg1;
        Double leg2;

        getHeight() : double {query}
        {

        }
```

```
getBase1() : double {query}
{

}

getBase2() : double {query}
{

}

getLeg1() : double {query}
{

}

getLeg2() : double {query}
{

}

setHeight (in newHeight : double)
{
     Height = newHeight
}



setBase1 (in newBase1Value : double)
{
     Base1 = newBase1Value
}

setBase2(in newBase2Value : double)
{
     Base2 = newBase2Value
}

setLeg1 (in newLeg1Value : double)
{
     Leg1 = newLeg1Value
}

setLeg2 (in newLeg2Value : double)
{
     Leg2 = newLeg2Value
```

```
      }

      calculateArea() : double {query}
      {
            Double temp = (½) * (base1 + base2) * (height)
      }

      calculatePerimeter() : double {query}
      {
            Double temp = base1 + base2 + leg1 + leg2
      }

      displayStatistics()
      {
            Write ("The height of the trapezoid is " + height)
            Write ("The bases of the trapezoid are" + base1 + " and "
+ base2)
            Write ("The legs of the trapezoid are" + leg1 + " and " +
leg2)
            Write("The area of the trapezoid is " +
calculatePerimeter())
      }
}
```

Ex. 14:

The following statements will return the array data after sorting the N elements that were in it. The sortedList temp is not returned in place of data because the sortedList might not have been implemented using arrays.

```
sortedList temp = createSortedList();
for(int index = 0; index < data.size(); index++)
{
      temp.sortedAdd(data[index]);
}
// data is now sorted within temp and has to be transferred back to
the original array
data.removeAll();
for(int j = 0; j < temp.sortedSize(); j++)
{
        Data[j] = temp.sortedGet(j);
}
Return data;
```

Chapter 5: 2,3,7,9,12,14,19 (pages 303-307)

Ex. 2:

a)

```
    Public void insert(Node n)
    {
        N.next = head;
        Head = n;
    }

    Public Node delete()
    {
        Node temp = head;
        Head = temp.next;
        Return temp;
    }
```

b)

```
    Public void endInsert(Node n)
    {
        Node curr = head
        while(curr.next != null)
        {
            Curr = curr.next;
        }
        curr.next = n;
    }

    Public void endDelete()
    {
        Node curr = head
        while(curr.next.next != null)
        {
            Curr = curr.next;
        }
        Node temp = curr.next;
        curr.next = null;
        Return temp;
    }
```
c)
```
        Public Node tailDelete()
        {
            Node curr = head
            while(curr.next != tail)
            {
                Curr = curr.next;
            }
            Node temp = tail;
            curr.next = null;
            Tail = curr;
            Return temp;
        }
```

Ex. 3:

a)
```
        Public static Object randomlyRemove()
        {
            if(aList.isEmpty())
            {
                Return null;
            }
            Random rand = new Random();
            Int randomIndex = rand.nextInt(aList.size());
            Object itemToBeRemoved = aList.get(randomIndex);
            aList.remove(randomIndex);
            Return itemToBeRemoved;
        }
```

b)
```
        Public Object randomRemove()
        {
```

```
        if(numItems =- 0)
        {
             Return null;
        }
        Random rand = new Random();
        Int randomIndex = rand.nextInt(numItems);
        Node current = head;
        for(int i = 0; i < randomIndex; i++)
        {
             Current = current.next;
        }
        Object itemToBeRemoved = current.item;
        if(randomIndex == 0)
        {
             Head = head.next;
        }
        else
        {
             Node previous = head;
             for(int j = -; j < randomIndex - 1; j++)
             {
                  Previous.next = current.next;
             }
        }
        numItems-;
        Return itemToBeRemoved;
   }
```

c)

```
   Public Object randomRemove()
   {
        if(numItems == 0)
        {
             Return null;
        }
        Random rand = new Random();
        Int randomIndex = rand.nextInt(numItems);
        Object itemToBeRemoved = items[randomIndex];
        for(int i = randomIndex + 1; i <= size(); i++)
        {
             Items[i - 1] = items[i];
```

```
          }
          numItems–;
          Return itemToBeRemoved;
     }
```

Ex. 7:

```
Import java.util.Iterator;
Import java.util.LinkedList;
Public static LinkedList<Integer> merge (LinkedList<Integer> a,
LinkedList<Integer> b)
{
     LinkedList<Integer> result = new LinkedList<Integer>();
     Iterator<Integer> aI = a.iterator();
     Iterator<Integer> bI = b.iterator();
     Int aTemp = 0, bTemp = 0;
     // first values of both linked lists using next method
     aTemp = aI.next();
     bTemp = bI.next();
     while(aI.hasNext() && bI.hasNext())
     {
          if(aTemp > bTemp)
          {
               Result.add(aTemp);
               aTemp = aI.next();
          }
```

```
            Else
            {
                  result.add(bTemp);
                  bTemp = bI.next();
            }
      }
      // add the last element to the result list of iterator of
whose list is empty
      if(!aI.hasNext())
      {
            result.add(aTemp);
      }
      Else
      {
            result.add(bTemp);
      }
      // add remaining elements to the result list if any one of
lists is non-empty
      while(aI.hasNext())
      {
            result.add(aTemp);
            aTemp = aI.next();
      }
      while(bI.hasNext())
      {
            result.add(bTemp);
            bTemp = bI.next();
      }
      if(!aI.hasNext())
      {
            result.add(aTemp);
      }
      Else
      {
            result.add(bTemp);
      }
      Return result;
}
```

Ex 9:

```
Public int findAnItem(Item item)
{
     Node current = list.next;
     Int position = 0;
     while(list != current)
     {
          if(current.item == item)
          {
               Return position;
          }
          position++;
     }
     Return -1;
}
```

Ex 12:

a)

```
     Import java.util.LinkedList;
     Public class SantasList()
     {
          Public LinkedList<Kids> niceList = new
     LinkedList<Kids>();
          Public LinkedList<Kids> naughtyList = new
     LinkedList<Kids>();

          Public void addKid(Kids k)
          {
               if(k.type.equals("nice"))
               {
                    niceList.add(k);
               }
               Else if(k.type.equals("naughty"))
               {
                    naughtyList.add(k);
               }
          }

          Public void printNice()
          {
               System.out.println("Nice list:");
               for(int i = 0; i < niceList.size(); i++)
               {
                    Kids temp = niceList.get(i);
                    System.out. println(i + ". " + temp.name);
               }
          }

          Public void printNaughty()
          {
               System.out.println("Naughty list");
               for(int i = 0; i < naughtyList.size(); i++)
               {
                    Kids temp = naughtyList.get(i);
                    System.out.println(i + ". " + temp.name);
               }
          }
     }
```
b)

```
          Public ListReferenceBased niceList = new
     ListReferenceBased();
          Public ListReferenceBased naughtyList = new
     ListReferenceBased();
          Public void addKid(Kids k)
          {
               if(k.type.equals("nice"))
               {
                    niceList.add(niceList.size() + 1, k);
               }
               Else if(k.type.equals("naughty"))
               {
                    naughtyList.add(naughtyList.size() + 1, k)
               }
          }
```

```
    Public void printNice()
    {
         System.out.println("Nice list:");
         for(int i = 0; i < niceList.size(); i++)
         {
              Kids temp = niceList.get(i);
              System.out. println(i + ". " + temp.name);
         }
    }
    Public void printNaughty()
    {
         System.out.println("Naughty list");
         for(int i = 0; i < naughtyList.size(); i++)
         {
              Kids temp = naughtyList.get(i);
              System.out.println(i + ". " + temp.name);
         }
    }
}
```

c) Nothing will have to be changed from part b if we wanted to use ListArrayBased class instead of the ListReferenceBased class. This is due to the fact that they both implement the same ListInterface, making their method signatures the exact same.

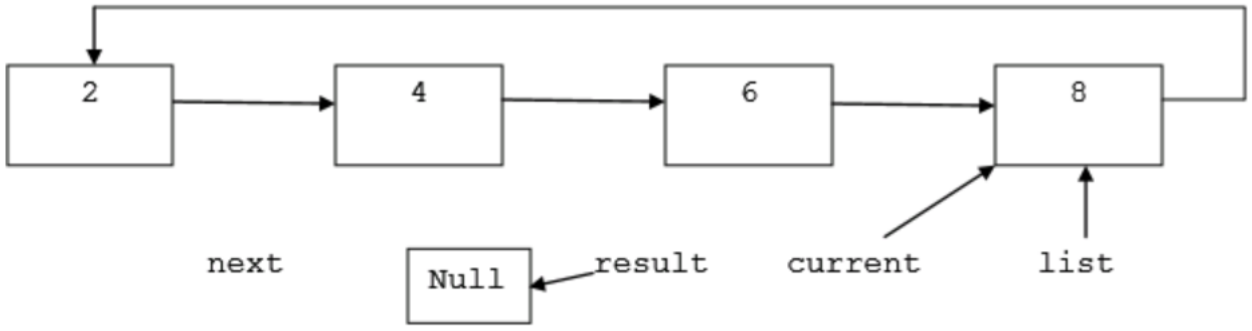
Ex 14:

```
Public void reverseList(Node list)
{
     Node result = null;
     Node current = list;
     Node next;
     while(current != null)
     {
          Next = current.next;
          Current.next = result;
          Current = next;
          if(current == list)
          {
               break;
          }
```

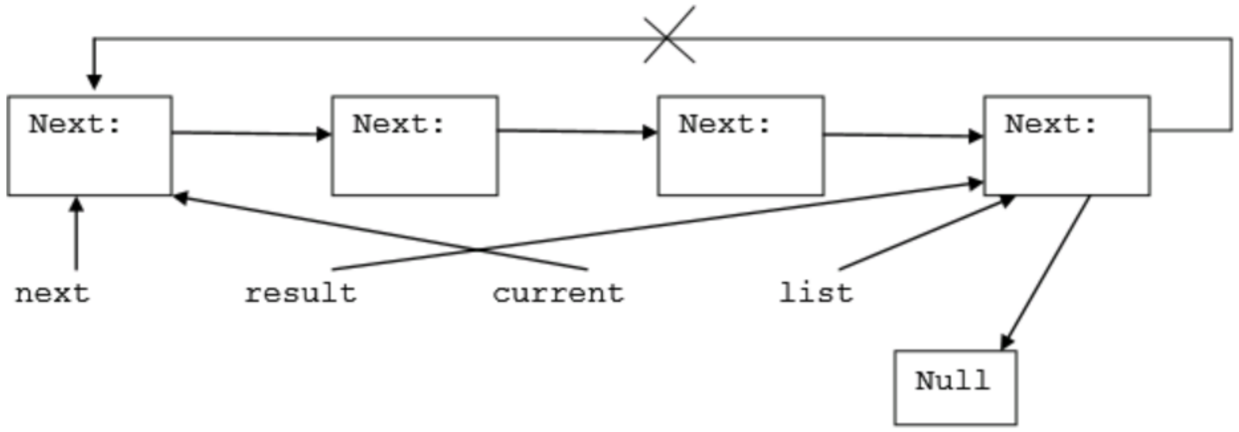
```
        }
        List.next = result;
        while(!(result.next == current))
        {
             Result = result.next;
        }
        List = result;
}
```

Below is a diagram illustrating how the method works:

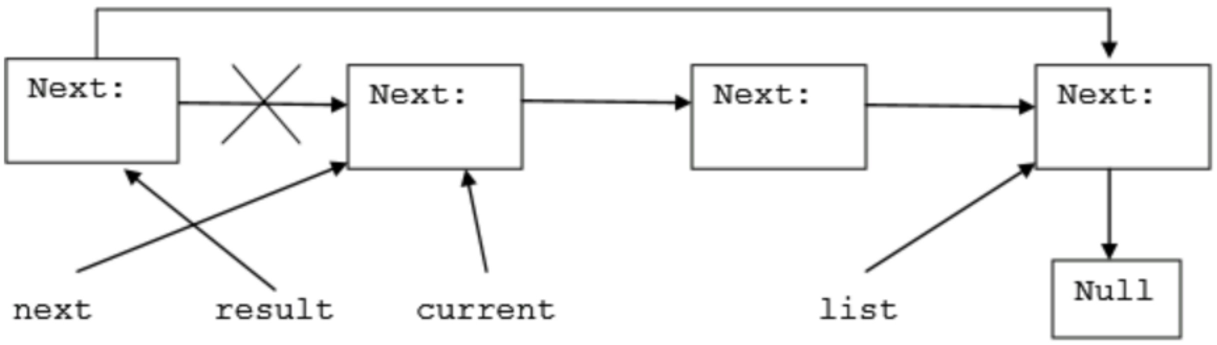
This is the list before the loop runs:



After iteration 1:

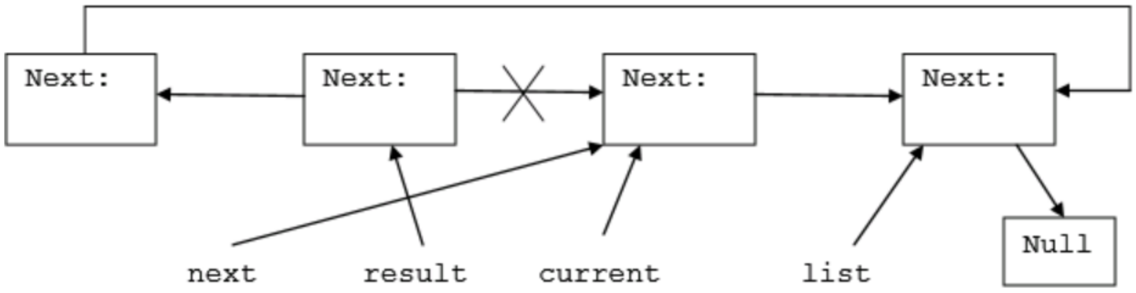


After iteration 2:

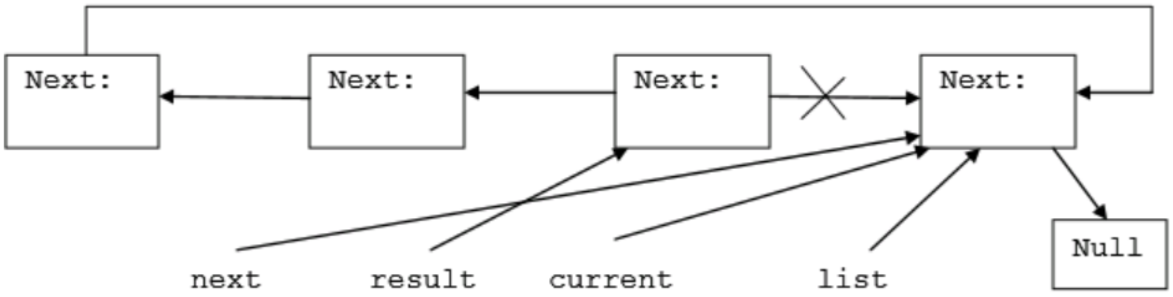


After iteration 3:

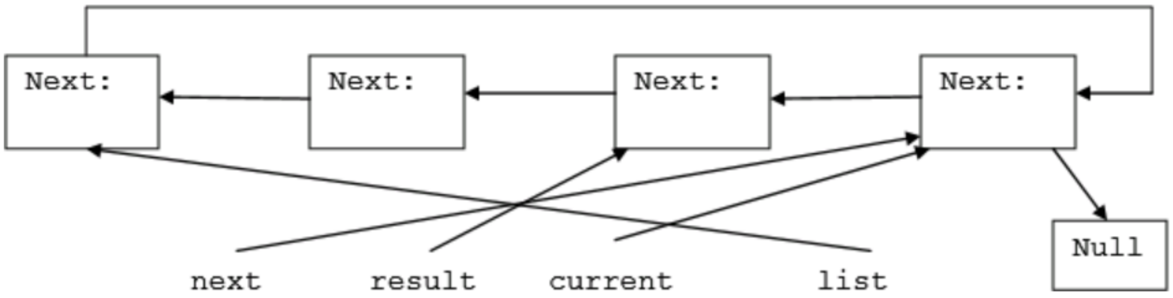
Step 2/2



After iteration 4:



The loop breaks after iteration 4 since `current = list` and the last statements run.



Ex 19:

a)

```
Vector<Integer> first = new Vector<Integer>(10,5);
for(int i = 1; i <= 10; i++)
{
     first.add(i);
}
```

b)

```
first.add(11);
first.add(12);
first.add(13);
     first.add(14);
```

The size() method now returns 14 and the capacity() methods now return 15.

c)

```
for(int j = 0; j < first.size(); j++)
{
     System.out.println(first.get(j));
}
```

The result of this loop is:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

d)

```
Vector<Integer> second = new Vector<Integer>();
```

When the capacity() method is called on the second vector, the number 10 is returned.

The following loop is used to add 11 elements into the vector second since the initial capacity was found to be 10.

```
for(int i = 0; i <= 10; i++)
{
     second.add(i);
}
```

The new capacity after adding 11 elements is 20.

e)

All of the elements in the vector second are deleted using the method clear(). The capacity of the second after the clear() method is still 20. The vector class behaved this way because it is expensive to deallocate memory when clearing a list. A user can delete a vector by removing all pointers to it and the Java garbage collector will mark it for deletion. However if a user calls the clear() method instead of setting the reference for the vector to null, it can be assumed that the vector will be reused again.

Chapter 7: 3,8,9,11,14 (pages 395-398)

Ex 3:

The following displayContentsOfStackInReverseOrder method accepts a stack aStack as a parameter and then displays the contents of aStack in reverse order with the help of an empty auxiliary stack auxStack.

a)

```
public static void
displayContentsOfStackInReverseOrder(StackReferenceBased aStack)

{

     StackReferenceBased auxStack = new StackReferenceBased();

     Object item;

     // move all the items of aStack to auxStack

     while(!aStack.isEmpty())

     {

          item = aStack.pop();

          auxStack.push(item);

     }

     // move all the items of auxStack to aStack

     while(!auxStack.isEmpty())

     {

          item = auxStack.pop();

          System.out.println(item);

          aStack.push(item);

     }
}//end of displayContentsOfStackInReverseOrder method
```

b.

The following countNumberOfItemInStack method accepts a stack aStack as a parameter and then counts and returns the number of items in the aStack with the help of an empty auxiliary stack auxStack.

```
// countNumberOfItemInStack method implementation

public static int countNumberOfItemInStack(StackReferenceBased
aStack)

{

     StackReferenceBased auxStack =

     new StackReferenceBased();

     Object item;

     int count = 0;

     while(!aStack.isEmpty())

     {

          // remove and get the top item of aStack

          item = aStack.pop();

          // push the top item of aStack into auxStack

          auxStack.push(item);

          // increment the number of items in the

          // aStack by 1

          Count++;

     }

     while(!auxStack.isEmpty())
```

```
    {
        // remove and get the top item of auxStack
        item = auxStack.pop();
        // push the top item of auxStack into aStack
        aStack.push(item);
    }
    // return the number of items in the aStack
    return count;
} // end of countNumberOfItemInStack method
```

c.

The following deleteAllOccurrencesOfStackItem method accepts a stack aStack and an item to be deleted as two parameters and then deletes all occurrences of the given item from aStack with the help of an empty auxiliary stack auxStack.

public static void deleteAllOccurrencesOfStackItem(StackReferenceBased aStack, Object itemToBeDeleted)

```
{
    // create an empty auxiliary stack auxStack
    StackReferenceBased auxStack =
    new StackReferenceBased();
    Object item;
    while(!aStack.isEmpty())
    {
        // remove and get the top item of aStack
```

```
            item = aStack.pop();

            // verify whether the top item is equal to

            // the given item

            if(!item.equals(itemToBeDeleted))

            {

            // push the top item of aStack into

            // auxStack

            auxStack.push(item);

            }

       }

       // move all the items of auxStack to aStack

       while(!auxStack.isEmpty())

       {

            // remove and get the top item of auxStack

            item = auxStack.pop();

            // push the top item of auxStack into aStack

            aStack.push(item);

       }

  } // end of deleteAllOccurrencesOfStackItem method
```

Program to test the above three tasks:

File: TestStack.java

```
// TestStack class implementation
public class TestStack
{
    // a)
    // displayContentsOfStackInReverseOrder method accepts
    // a stack aStack as a parameter and then displays the
    // contents of aStack in reverse order with the help
    // of an empty auxiliary stack auxStack
    public static void
    displayContentsOfStackInReverseOrder(StackReferenceBased
    aStack)
    {
        // create an empty auxiliary stack auxStack
        StackReferenceBased auxStack =
        new StackReferenceBased();
        Object item;
        // move all the items of aStack to auxStack
        while(!aStack.isEmpty())
        {
            // remove and get the top item of aStack
            item = aStack.pop();
            // push the top item of aStack into auxStack
```

```
          auxStack.push(item);
     }
// move all the items of auxStack to aStack
// and display each item
     while(!auxStack.isEmpty())
     {
          // remove and get the top item of auxStack
          item = auxStack.pop();
          // display the top item
          System.out.println(item);
          // push the top item of auxStack into aStack
          aStack.push(item);
     }
}//end of displayContentsOfStackInReverseOrder method
// b)
// countNumberOfItemInStack method accepts a stack
// aStack as a parameter and then counts and returns
// the number of items in the aStack with the help of
// an empty auxiliary stack auxStack
public static int
countNumberOfItemInStack(StackReferenceBased aStack)
{
     // create an empty auxiliary stack auxStack
```

```
StackReferenceBased auxStack =
new StackReferenceBased();
Object item;
int count = 0;
// move all the items of aStack to auxStack
// and count the number of items in aStack
while(!aStack.isEmpty())
{
     // remove and get the top item of aStack
     item = aStack.pop();
     // push the top item of aStack into auxStack
     auxStack.push(item);
     // increment the number of items in the
     // aStack by 1
     Count++;
}
// move all the items of auxStack to aStack
while(!auxStack.isEmpty())
{
     // remove and get the top item of auxStack
     item = auxStack.pop();
     // push the top item of auxStack into aStack
     aStack.push(item);
```

```
        }

        // return the number of items in the aStack

        return count;

    } // end of countNumberOfItemInStack method


    // c)

    // deleteAllOccurrencesOfStackItem method accepts a

    // stack aStack and an item to be deleted as two

    // parameters and then deletes all occurrences of the //
    given item from the aStack with the help of an

    // empty auxiliary stack auxStack

    public static void
    deleteAllOccurrencesOfStackItem(StackReferenceBased aStack,
    Object itemToBeDeleted)

    {

        // create an empty auxiliary stack auxStack

        StackReferenceBased auxStack =

        new StackReferenceBased();

        Object item;

        // move all the items of aStack to auxStack

        // except the given item to be removed

        while(!aStack.isEmpty())

        {

        // remove and get the top item of aStack

        item = aStack.pop();
```

```
          // verify whether the top item is equal to
          // the given item
               if(!item.equals(itemToBeDeleted))
               {
               // push the top item of aStack into
               // auxStack
               auxStack.push(item);
               }
     }
     // move all the items of auxStack to aStack
          while(!auxStack.isEmpty())
          {
               // remove and get the top item of auxStack
               item = auxStack.pop();
               // push the top item of auxStack into aStack
               aStack.push(item);
          }
     } // end of deleteAllOccurrencesOfStackItem method
```

```
     public static void main(String[] args)
     {
          // create a stack aStack
          StackReferenceBased aStack =
          new StackReferenceBased();
```

```
// push several items into aStack
aStack.push(1);
aStack.push(2);
aStack.push(3);
aStack.push(4);
aStack.push(2);
aStack.push(3);
aStack.push(2);
// call the displayContentsOfStackInReverseOrder
// method
System.out.println(
"Contents of aStack in reverse order:");
displayContentsOfStackInReverseOrder(aStack);
// call the countNumberOfItemInStack method
System.out.print("\nNumber of item in aStack: "
+ countNumberOfItemInStack(aStack));
// call the deleteAllOccurrencesOfStackItem
// method
System.out.println("\n\nContents of aStack after
deleting all occurrences of 2:");
deleteAllOccurrencesOfStackItem(aStack, 2);
// display the final aStack
while(!aStack.isEmpty())
```

```
          {
               System.out.println(aStack.pop());
          }
     } // end of main method
} // end of TestStack class
```

Sample Output:

```
Contents of aStack in reverse order:

1

2

3

4

2

3

2

Number of item in aStack: 7

Contents of aStack after deleting all occurrences of 2:

3

4

3

1
```

Ex 9:

The mystery method appears to convert an integer n to its octal representation by repeatedly dividing it by 8 and pushing the remainder onto a stack. It then pops the elements from the stack and prints them out in reverse order, which produces the octal representation of n.

For n = 15, the output would be:
```
1
7
```
For n = 65, the output would be:
```
1
1
1
```

Explanation for n = 15:

- The first while loop pushes the remainders of 15 / 8 = 1 and 1 / 8 = 0 onto the stack.
- The second while loop pops these values from the stack and prints them out in reverse order, which produces the octal representation 17.

Explanation for n = 65:

- The first while loop pushes the remainders of 65 / 8 = 1, 8 / 8 = 0, and 1 / 8 = 0 onto the stack.
- The second while loop pops these values from the stack and prints them out in reverse order, which produces the octal representation 101.

Ex 14:

a. To determine whether a string is in the language L = [w:w contains equal numbers of A's and B's in any order}, we can use a stack to keep track of the number of A's and B's in the string. The pseudocode for the method in Java is as follows:

```
public boolean isInLanguage(String s) {

    Stack<Character> stack = new Stack<Character>();

    int aCount = 0;

    int bCount = 0;

    for (int i = 0; i < s.length(); i++) {

        char c = s.charAt(i);

        if (c == 'A') {

            aCount++;

            stack.push(c);

        } else if (c == 'B') {

            bCount++;

            if (stack.empty() || stack.peek() != 'A') {

                return false;

            } else {

                stack.pop();

            }

        }

    }

    return aCount == bCount && stack.empty();
}
```

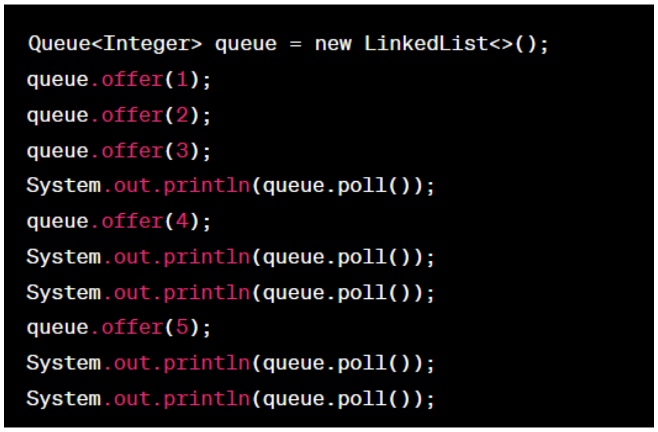
b. To determine whether a string is in the language L = {w:w is of the form A^2nB^n for some n>=0}, we can use a stack to keep track of the number of A's in the string. The pseudocode for the method in Java is as follows:

```
public boolean isInLanguage(String s) {

    Stack<Character> stack = new Stack<Character>();



    for (int i = 0; i < s.length(); i++) {

        char c = s.charAt(i);

        if (c == 'A') {

            stack.push(c);

        } else if (c == 'B') {

            if (stack.empty()) {

                return false;

            } else {

                stack.pop();

            }

        } else {

            return false; // the string contains an invalid
character

        }

    }



    return stack.empty();

}
```

Chapter 8: 3,5,6,7,8,13,14 (pages 446-448)

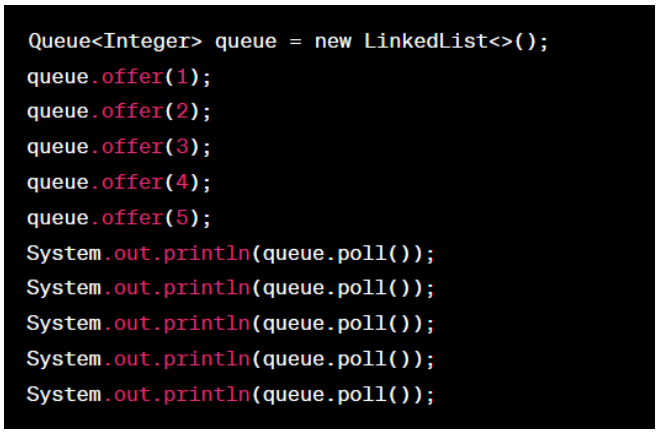
Ex 3:

a) To produce the sequence 1 3 5 4 2, the following operations should work.



```java
Queue<Integer> queue = new LinkedList<>();
queue.offer(1);
queue.offer(2);
queue.offer(3);
System.out.println(queue.poll());
queue.offer(4);
System.out.println(queue.poll());
System.out.println(queue.poll());
queue.offer(5);
System.out.println(queue.poll());
System.out.println(queue.poll());
```

b) To produce 1 2 3 4 5, the following operations should work:



```java
Queue<Integer> queue = new LinkedList<>();
queue.offer(1);
queue.offer(2);
queue.offer(3);
queue.offer(4);
queue.offer(5);
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll());
```

c) The sequence 1 2 3 4 5 is always possible because the values are enqueued in the correct order and dequeued in the correct order. However, not all sequences of length 5 are possible. For example, the sequence 2 1 3 5 4 is not possible because once 2 is enqueued, it must be dequeued before 1 can be dequeued. Similarly, once 5 is enqueued, it must be dequeued before 4 can be dequeued. Therefore, the sequence 2 1 3 5 4 cannot occur.

Ex 6:

```
import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;

public class LanguageRecognition {

    public static boolean recognizeLanguage(String inputString) {

        Queue<Character> queue = new LinkedList<Character>();

        Stack<Character> stack = new Stack<Character>();

        boolean foundDollar = false;

        int index = 0;

        while (index < inputString.length()) {

            char currentChar = inputString.charAt(index);

            if (currentChar == '$') {

                foundDollar = true;

                index++;

                continue;

            }

            if (!foundDollar) {

                queue.offer(currentChar); // Add to the end of the queue

            } else {

                stack.push(currentChar); // Add to the top of the stack

            }

            index++;
        }

        while (!queue.isEmpty() && !stack.isEmpty()) {

            if (queue.poll() != stack.pop()) {

                return false; // Mismatch found

            }

        }

        if (!queue.isEmpty() || !stack.isEmpty()) {

            return false; // One of the queue or stack is not empty,
hence not matching

        }

        return true;

    }

    public static void main(String[] args) {

        String inputString = "abc$cbadf";

        boolean result = recognizeLanguage(inputString);

        System.out.println("Does the input string \"" + inputString + "\"
belong to the language L? " + result);

    }

}
```

Ex 7:

This code segment represents a simulation of the Josephus problem using a queue data structure. The output of the code will be as follows:

```
3

-1

5

-7

1
```

Explanation:

- In the first for loop, the enqueue() method is called five times to add integers 1 to 5 into the queue.
- In the second for loop, two integers are dequeued from the queue, and their sum and difference are calculated and enqueued back into the queue.
    - On the first iteration of the loop, the integers 1 and 2 are dequeued. Their sum is 3, and their difference is -1. These values are enqueued back into the queue.
    - On the second iteration, the integers 3 and 2 are dequeued. Their sum is 5, and their difference is -1. These values are enqueued back into the queue.
    - On the third iteration, the integers 3 and 5 are dequeued. Their sum is 8, and their difference is -2. These values are enqueued back into the queue.
    - On the fourth iteration, the integers 8 and -2 are dequeued. Their sum is 6, and their difference is 10. These values are enqueued back into the queue.
    - On the fifth and final iteration, the integers 6 and 10 are dequeued. Their sum is 16, and their difference is -4. These values are enqueued back into the queue.
- In the while loop, all remaining elements in the queue are dequeued and printed to the console. The order of the printed values is determined by the order of enqueuing in the second for loop.

Ex 8:

The given code fragment reverses the order of elements in the queue q by using a stack s. Here's how it works:

1. A new stack s is created.
2. A loop is started to dequeue each element from the queue q and push it onto the stack s. This effectively reverses the order of the elements, as the first element dequeued from q will be at the bottom of the stack s, and the last element dequeued from q will be at the top of the stack s.
3. Another loop is started to pop each element from the stack s and enqueue it back onto the queue q. This will restore the original order of the elements in q, but with the first element that was originally dequeued now being at the back of the queue, and the last element that was originally dequeued now being at the front of the queue.

So, the code essentially swaps the order of the elements in the queue by temporarily using a stack as an intermediate data structure.

Ex 14:

```
public static Object getLastElement(QueueInterface queue) throws
QueueException {
    if (queue.isEmpty()) {
        throw new QueueException("Cannot get last element of an
empty queue.");
    }
    QueueInterface newQueue;
    if (queue instanceof QueueArrayBased) {
        newQueue = new QueueArrayBased();
    } else {
        newQueue = new QueueListBased();
    }
    Object lastElement = null;
    while (!queue.isEmpty()) {
        Object element = queue.dequeue();
        newQueue.enqueue(element);
        lastElement = element;
    }
    while (!newQueue.isEmpty()) {
        Object element = newQueue.dequeue();
        queue.enqueue(element);
    }
    return lastElement;
}
```

Chapter 3: 4,10,11,12,13,14,15,18 (pages 189-193)

Ex 4:

```
public static int sum(int start, int end) {
    if (start == end) {
        return start;
    } else {
        return start + sum(start + 1, end);
    }
}
```

Here's how this method works:

- If start and end are the same, then we just return that number, since the sum of a single number is itself.

- Otherwise, we add start to the result of calling sum with start + 1 as the new starting value and end as the ending value. This effectively reduces the problem to summing the range from start + 1 to end, which we can solve recursively until we reach the base case.

Here's an example of how to call the sum method:

```
int start = 1;

int end = 5;

int sum = sum(start, end); // sum is now 15
```

In this example, the sum method will be called with start = 1 and end = 5, which will result in the sum of the integers from 1 to 5 (inclusive) being calculated and returned as the result.

Ex 10:

a)

```
public static void writeLine(char c, int n) {
    if (n == 0) {
        return;
    }
    System.out.print(c);
    writeLine(c, n-1);
}
```

b)

```
public static void writeBlock(char c, int n, int m) {
    if (m == 0) {
        return;
    }
    writeLine(c, n);
    System.out.println();
    writeBlock(c, n, m-1);
}
```

Ex 11:

The program will produce the following output:

```
z(2) = 2 * z{1) + 1
z(1) = 4
z(0) = 2
42
```

The guess() method is a recursive function that takes an integer array c and an integer x as input parameters. The method computes the value of z(n) based on the values of z(n-1) and z(n-2), where z(0) = c[0] and z(1) = x * z(0) + c[1]. The method recursively calls itself until the base case, where the length of the array c is 1.

In the main() method, an integer array x is created with the values {2, 4, 1}. The guess() method is then called with x and 5 as parameters.

The output shows the values of z(2), z(1), and z(0) computed by the guess() method, followed by the final output value, which is the result of the guess() method with x and 5 as parameters, which is 42.

Ex 12:

The program implements binary search to find the integer square root of the input number n. It recursively divides the search range in half until it finds the square root. Here is what the output of the program would look like when run for n = 64:

```
Enter: a = 1, b = 64, mid = 32
Enter: a = 33, b = 64, mid = 48
Enter: a = 33, b = 47, mid = 40
Enter: a = 33, b = 39, mid = 36
Enter: a = 37, b = 39, mid = 38
Leave: a = 37, b = 39, mid = 38
Leave: a = 33, b = 39, mid = 36
Leave: a = 33, b = 47, mid = 40
Leave: a = 33, b = 64, mid = 48
Leave: a = 1, b = 64, mid = 32

For n = 64, the result is 8
```

The program finds the square root of 64 to be 8. The search range starts with a = 1 and b = n, which are 1 and 64 respectively in this case. The mid-point of the range is calculated as (a + b) / 2, which is 32 for the first iteration. The program checks if mid * mid is less than or equal to n and if (mid + 1) * (mid + 1) is greater than n. If true, it returns mid as the square root. If mid * mid is greater than n, it recursively searches the lower half of the range (a to mid - 1). Otherwise, it searches the upper half (mid + 1 to b). The program prints the values of a, b, and mid as it enters and leaves each call to search().

Ex 13:

a)Tracing with n = 88;

```
displayOctal(88)
     displayOctal(11)
```

```
          displayOctal(1)
               System.out.println(1)
          System.out.println(3)
     System.out.println(0)
     System.out.println(1)
```

Therefore, the octal representation of the decimal number 88 is 130.

b) The four questions for constructing a recursive solution are:

1. What is the simplest possible input?
    - The simplest possible input is when n is less than or equal to 0.
2. How does each recursive call simplify the problem?
    - Each recursive call divides n by 8, which shifts the decimal point to the left by one digit and effectively removes the last digit from the number.
3. What is the base case?
    - The base case is when n is less than or equal to 0, in which case the method does nothing.
4. What value is returned or what effect is produced when the base case is reached?
    - When the base case is reached, no value is returned and no effect is produced. The recursion simply ends.

In this method, the base case is handled implicitly by the if statement at the beginning of the method. If n is less than or equal to 0, the method does nothing. Each recursive call simplifies the problem by dividing n by 8. Finally, the solution is constructed by printing the remainder of n when divided by 8 at each step.

Ex 14:

The output of the program will be:

```
Enter f: n = 8
Enter f: n = 7
Enter f: n = 5
Enter f: n = 4
Enter f: n = 2

f(8) is equal to 30
```

The program starts by calling the f method with the argument 8. Inside the f method, the switch statement is used to determine what value to return based on the value of n. Since n is 8, the default case is executed, which calls the f method again with n-1 (7) and n-3 (5) as arguments, and multiplies their results.

The f method is called again with the argument 7, and the same process happens until the argument passed is 2. When n is 2, the case 2 is executed, and the method returns n+1 (3). This value is then used to calculate the result of f(4), f(5), f(7), and finally f(8).

Therefore, the output of the program is "f(8) is equal to 30".

To cause the program to run forever, you could pass a negative value as an argument to the f method. This would cause an infinite recursive loop because the default case will keep calling the f method with decreasing values of n, which will never reach 1, 2, or 3 to stop the recursion.

Ex 15:

If we execute the method with x = 5 and y = 3, it will print the following output:

```
6 2
7 1
8 0
8 0
```

```
7 1
6 2
```

Here's what's happening in the code:

1. The method is called with x = 5 and y = 3.
2. Since y is greater than 0, x is incremented to 6 and y is decremented to 2.
3. The values of x and y are printed as "6 2".
4. The method is called recursively with the new values of x and y.
5. Since y is still greater than 0, x is incremented to 7 and y is decremented to 1.
6. The values of x and y are printed as "7 1".
7. The method is called recursively with the new values of x and y.
8. Since y is still greater than 0, x is incremented to 8 and y is decremented to 0.
9. The values of x and y are printed as "8 0".
10. Since y is now equal to 0, the recursion stops and control returns to the previous call of the method.
11. The values of x and y are printed again as "8 0".
12. Control returns to the initial call of the method.
13. The values of x and y are printed again as "7 1".
14. Control returns to the initial call of the method.
15. The values of x and y are printed again as "6 2".
16. The recursion stops and the method completes.

Ex 18:

a)

```
def power1(x, n):
    result = 1
    for i in range(n):
        result *= x
    return result
```

b)
```
def power2(x, n):
    if n == 0:
        return 1
    else:
        return x * power2(x, n-1)
```

c)
```
def power3(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        y = power3(x, n/2)
        return y*y
    else:
        y = power3(x, (n-1)/2)
        return x*y*y
```

d) To compute 3^32, each method will perform the following number of multiplications:

- power1: 31 multiplications
- power2: 31 multiplications
- power3: 12 multiplications

To compute 3^19, each method will perform the following number of multiplications:

- power1: 18 multiplications
- power2: 18 multiplications
- power3: 8 multiplications

e) To compute 3^32, power2 will make 32 recursive calls, and power3 will make 5 recursive calls. To compute 3^19, power2 will make 19 recursive calls, and power3 will make 4 recursive calls.

Chapter 10: 1,5,6,7,11,12,13,14,15,18,20,23 (pages 554-557)

Ex 1:

a) Computing the sum of the first half of an array of n items:

In the worst case, we need to iterate over half of the n items in the array, so the time complexity is O(n/2). However, we can simplify this to O(n) since in big O notation, we drop the constant factor.

b) Initializing each element of an array items to 1: In the worst case, we need to iterate over all n elements in the array and assign each one to 1. Therefore, the time complexity is O(n).

c) Displaying every other integer in a linked list of n nodes: In the worst case, we need to iterate over all n nodes in the linked list and print out every other one. This gives a time complexity of O(n).

d) Displaying all n names in a circular linked list: In the worst case, we need to iterate over all n nodes in the circular linked list to display all n names. Therefore, the time complexity is O(n).

e) Displaying the third element in a linked list: Since we only need to access a specific element, this can be done in constant time, giving a time complexity of O(1).

f) Displaying the last integer in a linked list of n nodes: In the worst case, we need to iterate over all n nodes in the linked list to reach the last one. This gives a time complexity of O(n).

g) Searching an array of n integers for a particular value by using a binary search: Binary search has a time complexity of O(log n) in the worst case. This is because with each comparison, we can eliminate half of the remaining search space, resulting in a logarithmic time complexity.
h) The worst-case time complexity of mergesort algorithm for sorting an array of n integers is O(n*log(n)).

Ex 5:
The order of the algorithm is O(n^2).
The first loop: for(int pass = 1; pass <= n; ++pass) will perform n comparisons.
The second loop: for(int index = 0; index < n; ++index) will

perform n comparisons.
The third loop: for(int count = 1; count < 10; ++count) will perform 9 comparisons.
As the computations are independent of n, the order of the algorithm is n*n*9 = O(n^2)

Ex 6:

```
Public static void f(int[] theArray, int n)
{
      Int temp;
      // TOTAL # OF ITERATIONS = N - 1
      // TOTAL # OF COMPARISONS = N
      for(int j =0; j < n; ++j)
      {
            Int i = 0;
            // 1ST ITERATION OF OUTER LOOP
            // # OF ITERATIONS = 1, # OF COMPARISONS = 2
            // 2ND ITERATION OF OUTER LOOP
            // # OF ITERATIONS = 2, # OF COMPARISONS = 3
            // 3RD ITERATION OF OUTER LOOP
            // # OF ITERATIONS = 3, # OF COMPARISONS = 4
            // (N-1) ITERATION OF OUTER LOOP
            // # OF ITERATIONS = N - 1, # OF COMPARISONS = 4
            While (i <= j)
            {
                  // # OF COMPARISONS = 1 IN EACH ITERATION
                  if(theArray[i] < (theArray[j]))
                  {
                        Temp = theArray[i];
                        theArray[i] = theArray[j];
                        theArray[j] = temp;
                  }
                  ++i;
            }
      }
}
```

The total # of comparisons for the outer loop and the inner loop is computed below:
The # of comps for the iterator of the outer loop is equal to n.
The formula for the sum of first n natural numbers is (n(n-1))/2, where n is the value of the last element in the sequence.
The number of comps for the iterator of the inner loop is as follows:
2 + 3 + 4 + 5 + … + n
= 1 + 2 + 3 + 4 + … +(n-1)

= (n-1)(n+1-1)/2
= n(n-1)/2
The # of comps for the if condition in the inner loop is as follows:
1 + 2 + 3 + 4 + … + n
= n(n+1)/2
Total # of comps:
N + (n(n-1)/2))+(n(n+1)/2)
= 2n + n^2 - n + n^2 + n/2
= 2n + 2n^2/2
= 2(n^2 + n)/2
= n(n+1)

Ex 7:

No, in general, selection sort is not faster than insertion sort for large arrays, even in the worst case. Both algorithms have worst-case time complexity of O(n^2), where n is the size of the array. However, insertion sort tends to perform better than selection sort in practice for most inputs, particularly for small or partially sorted arrays.

The main reason for this is that insertion sort has better cache locality than selection sort. In insertion sort, elements are gradually inserted into their correct positions in the array, meaning that neighboring elements are more likely to be accessed together. This can result in better cache performance and faster execution time.

In contrast, selection sort scans the entire unsorted portion of the array to find the minimum element, and then swaps it with the first unsorted element. This can result in more random memory access patterns and less efficient cache usage.

Therefore, while selection sort may have a slightly better worst-case time complexity than insertion sort, in practice, insertion sort tends to be faster for most inputs, particularly for small or partially sorted arrays.

Ex 11:

Here are the steps of insertion sort as it sorts the array [80, 40, 25, 20, 30, 60] into ascending order:

1. We start with the original array: [80, 40, 25, 20, 30, 60]

2. We start at index 1, which is the second element of the array (40), and compare it to the element to its left (80). Since 40 is smaller than 80, we swap these two elements.
    - The array now looks like this: [40, 80, 25, 20, 30, 60]
3. We move to index 2 (25) and compare it to the elements to its left (80 and 40). 25 is smaller than both, so we swap it with 40, then swap it with 80.
    - The array now looks like this: [25, 40, 80, 20, 30, 60]
4. We move to index 3 (20) and compare it to the elements to its left (80, 40, and 25). 20 is smaller than all three, so we swap it with 25, then with 40, and finally with 80.
    - The array now looks like this: [20, 25, 40, 80, 30, 60]
5. We move to index 4 (30) and compare it to the elements to its left (80, 40, 25, and 20). 30 is larger than 20 and 25, but smaller than 40 and 80. We insert it in the correct position between 25 and 40 by swapping it with 40 and then with 25.
    - The array now looks like this: [20, 30, 25, 40, 80, 60]
6. We move to index 5 (60) and compare it to the elements to its left (80, 40, 25, 30, and 20). 60 is larger than all of them, so we leave it in place.
    - The array is now sorted in ascending order: [20, 25, 30, 40, 60, 80]

So the sorted array is [20, 25, 30, 40, 60, 80], and it took 5 passes through the array to sort it using insertion sort.

Ex 12:

Here are the steps of selection sort on the array [8, 11, 23, 1, 20, 33]:
Starting with the first element, which is 8, compare it with all the other elements in the array to find the smallest value. The smallest value is 1, which is in the 4th position.
[8, 11, 23, 1, 20, 33]

1. Swap the first element (8) with the smallest value found in the array (1).
   [1, 11, 23, 8, 20, 33]

Move to the second element (11) and repeat the process of finding the smallest value in the unsorted portion of the array.

The smallest value is 8, which is in the 4th position.
[1, 11, 23, 8, 20, 33]

2. Swap the second element (11) with the smallest value found in the array (8).
   [1, 8, 23, 11, 20, 33]

Move to the third element (23) and repeat the process of finding the smallest value in the unsorted portion of the array. The smallest value is 11, which is in the 4th position.
[1, 8, 23, 11, 20, 33]

3. Swap the third element (23) with the smallest value found in the array (11).
   [1, 8, 11, 23, 20, 33]

Move to the fourth element (23) and repeat the process of finding the smallest value in the unsorted portion of the array. The smallest value is 20, which is in the 5th position.
[1, 8, 11, 23, 20, 33]

4. Swap the fourth element (23) with the smallest value found in the array (20).
   [1, 8, 11, 20, 23, 33]
5. Move to the fifth element (23) and repeat the process of finding the smallest value in the unsorted portion of the array. The smallest value is 23, which is already in the correct position.
   [1, 8, 11, 20, 23, 33]
6. Move to the sixth and final element (33), which is already in the correct position.
   [1, 8, 11, 20, 23, 33]

Ex 13:

Here are the steps of bubble sort on the array [10, 12, 23, 34, 5]:

Pass 1:

- Compare the first and second elements (10 and 12), and swap them because 10 > 12.
- Compare the second and third elements (12 and 23), and swap them because 12 > 23.
- Compare the third and fourth elements (23 and 34), and leave them in place because 23 < 34.
- Compare the fourth and fifth elements (34 and 5), and swap them because 34 < 5.
  [10, 12, 23, 5, 34]

Pass 2:

- Compare the first and second elements (10 and 12), and leave them in place because 10 < 12.
- Compare the second and third elements (12 and 23), and swap them because 12 > 23.
- Compare the third and fourth elements (23 and 5), and swap them because 23 < 5.
- Compare the fourth and fifth elements (5 and 34), and leave them in place because 5 < 34.
  [10, 23, 5, 12, 34]

Pass 3:

- Compare the first and second elements (10 and 23), and leave them in place because 10 < 23.
- Compare the second and third elements (23 and 5), and swap them because 23 > 5.
- Compare the third and fourth elements (5 and 12), and leave them in place because 5 < 12.
- Compare the fourth and fifth elements (12 and 34), and swap them because 12 < 34.
  [23, 5, 10, 12, 34]

Pass 4:

- Compare the first and second elements (23 and 5), and swap them because 23 < 5.
- Compare the second and third elements (5 and 10), and swap them because 5 > 10.
- Compare the third and fourth elements (10 and 12), and leave them in place because 10 < 12.
- Compare the fourth and fifth elements (12 and 34), and leave them in place because 12 < 34.
  [5, 10, 23, 12, 34]

Pass 5:

- Compare the first and second elements (5 and 10), and leave them in place because 5 < 10.
- Compare the second and third elements (10 and 23), and leave them in place because 10 < 23.
- Compare the third and fourth elements (23 and 12), and swap them because 23 > 12.
- Compare the fourth and fifth elements (12 and 34), and leave them in place because 12 < 34.
  [5, 10, 12, 23, 34]

Ex 14:

a. Inverted array: 9 7 5 3 1

Selection Sort:

1. In selection sort, the algorithm repeatedly finds the minimum element from the unsorted part of the array and moves it to the beginning of the sorted part of the array.

First pass:

1. 9 7 5 3 1 --> minimum element is 1

1 7 5 3 9 --> swap the minimum element with the first element of the unsorted part

2. Second pass:
3. 1 7 5 3 9 --> minimum element is 3

1 3 5 7 9 --> swap the minimum element with the second element of the unsorted part

4. Third pass:
5. 1 3 5 7 9 --> minimum element is 5

1 3 5 7 9 --> swap the minimum element with the third element of the unsorted part

6. Fourth pass:
7. 1 3 5 7 9 --> minimum element is 7

8. 1 3 5 7 9 --> swap the minimum element with the fourth element of the unsorted part

Therefore, the sorted array using selection sort is: 1 3 5 7 9.

Bubble Sort:

2. In bubble sort, the algorithm repeatedly swaps adjacent elements if they are in the wrong order.

First pass:

9 7 5 3 1 --> 7 9 5 3 1 --> 7 5 9 3 1 --> 7 5 3 9 1 --> 7 5 3 1 9

1. Second pass:

7 5 3 1 9 --> 5 7 3 1 9 --> 5 3 7 1 9 --> 5 3 1 7 9

2. Third pass:
3. 5 3 1 7 9 --> 3 5 1 7 9

3 1 5 7 9

4. Fifth pass:
5. 1 3 5 7 9

Therefore, the sorted array using bubble sort is: 1 3 5 7 9.

Insertion Sort:

3. In insertion sort, the algorithm iterates through the unsorted part of the array and inserts each element into its correct position in the sorted part of the array.

First pass:

1. 9 7 5 3 1 --> 7 9 5 3 1
2. 5 7 9 3 1 --> 5 7 3 9 1
3. 3 5 7 9 1 --> 3 5 7 1 9
4. 1 3 5 7 9

Therefore, the sorted array using insertion sort is: 1 3 5 7 9.

b. Ordered array: 1 3 5 7 9

Selection Sort:

1. In selection sort, the algorithm repeatedly finds the minimum element from the unsorted part of the array and moves it to the beginning of the sorted part of the array.

First pass:

1. 1 3 5 7 9 --> minimum element is 1
2. 1 3 5 7 9 --> minimum element is 3
3. 1 3 5 7 9 --> minimum element is 5
4. 1 3 5 7 9 --> minimum element is 7
5. 1 3 5 7 9 —-> minimum element is 9

Ordered array: 1 3 5 7 9

Ex 15:

In bubble sort, the worst case scenario occurs when the input array is in reverse sorted order. The best case scenario occurs when the input array is already sorted.
a. Worst case:

For an array of 100 elements, in the worst case scenario, each pass of bubble sort will move the largest unsorted element to its correct position at the end of the array. In the first pass, we need to compare 99 pairs of adjacent elements, in the second pass 98 pairs, and so on. Therefore, the total number of comparisons required for bubble sort in the worst case is:

99 + 98 + 97 + ... + 2 + 1 = (99*100)/2 = 4950

So, in the worst case, we would need 4950 comparisons to sort an array of 100 elements using bubble sort.

b. Best case:

If the input array is already sorted, then bubble sort will only need to do one pass through the array to check that no swapping is necessary. Therefore, in the best case, the number of comparisons required for bubble sort would be:

99 (comparing pairs of adjacent elements)

So, in the best case, we would need 99 comparisons to sort an array of 100 elements using bubble sort.

Ex 18:

The mergesort algorithm first divides the input array into smaller subarrays, sorts them recursively, and then merges them to produce the final sorted array. Here is the trace of the mergesort algorithm for the given input array:

Initial array: [80, 40, 25, 20, 30, 60]

Call mergesort([80, 40, 25, 20, 30, 60])

- Call mergesort([80, 40, 25])

        - Call mergesort([80])
        - Call mergesort([40, 25])
            - Call mergesort([40])
            - Call mergesort([25])
            - Call merge([40], [25]) -> [25, 40]
        - Call merge([80], [25, 40]) -> [25, 40, 80]
    - Call mergesort([20, 30, 60])
        - Call mergesort([20])
        - Call mergesort([30, 60])
            - Call mergesort([30])
            - Call mergesort([60])
            - Call merge([30], [60]) -> [30, 60]
        - Call merge([20], [30, 60]) -> [20, 30, 60]
    - Call merge([25, 40, 80], [20, 30, 60]) -> [20, 25, 30, 40, 60, 80]

Final sorted array: [20, 25, 30, 40, 60, 80]

The calls to mergesort occur in the following order:

1. mergesort([80, 40, 25, 20, 30, 60])
2. mergesort([80, 40, 25])
3. mergesort([80])
4. mergesort([40, 25])
5. mergesort([40])
6. mergesort([25])
7. merge([40], [25])
8. merge([80], [25, 40])
9. mergesort([20, 30, 60])
10. mergesort([20])
11. mergesort([30, 60])
12. mergesort([30])
13. mergesort([60])
14. merge([30], [60])
15. merge([20], [30, 60])
16. merge([25, 40, 80], [20, 30, 60])

Ex 20:

The mergesort algorithm first divides the input array into smaller subarrays, sorts them recursively, and then merges them to produce the final sorted array. Here is the trace of the mergesort algorithm for the given input array:

Initial array: [80, 40, 25, 20, 30, 60]

Call mergesort([80, 40, 25, 20, 30, 60])

- Call mergesort([80, 40, 25])

        - Call mergesort([80])
        - Call mergesort([40, 25])
            - Call mergesort([40])
            - Call mergesort([25])
            - Call merge([40], [25]) -> [25, 40]
        - Call merge([80], [25, 40]) -> [25, 40, 80]
    - Call mergesort([20, 30, 60])
        - Call mergesort([20])
        - Call mergesort([30, 60])
            - Call mergesort([30])
            - Call mergesort([60])
            - Call merge([30], [60]) -> [30, 60]
        - Call merge([20], [30, 60]) -> [20, 30, 60]
    - Call merge([25, 40, 80], [20, 30, 60]) -> [20, 25, 30, 40, 60, 80]

Final sorted array: [20, 25, 30, 40, 60, 80]

The calls to mergesort occur in the following order:

1. mergesort([80, 40, 25, 20, 30, 60])
2. mergesort([80, 40, 25])
3. mergesort([80])
4. mergesort([40, 25])
5. mergesort([40])
6. mergesort([25])
7. merge([40], [25])
8. merge([80], [25, 40])
9. mergesort([20, 30, 60])
10. mergesort([20])
11. mergesort([30, 60])
12. mergesort([30])
13. mergesort([60])
14. merge([30], [60])
15. merge([20], [30, 60])
16. merge([25, 40, 80], [20, 30, 60])

Trace the quicksort algorithm as it sorts the following array into ascending order.
List the calls to quicksort and to partition in the order in which they occur.
80 40 25 20 30 60 15
The quicksort algorithm selects a pivot element, partitions the array around the pivot, and then recursively sorts the two subarrays formed by the partition. Here is the trace of the quicksort algorithm for the given input array:

Initial array: [80, 40, 25, 20, 30, 60, 15]

Call quicksort([80, 40, 25, 20, 30, 60, 15])

- Call partition([80, 40, 25, 20, 30, 60, 15], 0, 6)
    - Pivot element: 15
    - [15, 40, 25, 20, 30, 60, 80]
    - Return index of pivot: 0
- Call quicksort([15, 40, 25, 20, 30, 60], 0, 0)
    - Already sorted
- Call quicksort([40, 25, 20, 30, 60], 1, 5)
    - Call partition([40, 25, 20, 30, 60], 1, 5)
        - Pivot element: 60
        - [40, 60, 25, 20, 30, 80]
        - Return index of pivot: 1
    - Call quicksort([40, 25, 20, 30], 1, 0)
        - Already sorted
    - Call quicksort([25, 20, 30], 2, 3)
        - Call partition([25, 20, 30], 2, 3)
            - Pivot element: 30
            - [25, 20, 30]
            - Return index of pivot: 2
        - Call quicksort([25, 20], 2, 1)
            - Swap [25, 20]
            - Call partition([25, 20], 2, 1)
                - Pivot element: 20
                - [20, 25]
                - Return index of pivot: 0
            - Call quicksort([], 2, 1)
                - Already sorted
        - Call quicksort([], 3, 3)
            - Already sorted
- Call quicksort([], 6, 6)
    - Already sorted

Final **sorted array: [15, 20, 25, 30, 40, 60, 80]**

Ex 23:

Yes, modifying the partition algorithm to eliminate unnecessary swapping can change the order of the algorithm. Here's an example of how the partition algorithm can be modified:

```
public static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            if (i != j) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }

    if (i+1 != high) {
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;
    }

    return i+1;
}
```

In this modified algorithm, we check whether i has been incremented before swapping the array elements at positions i and j. If i and j are the same, the swap is unnecessary and we skip it.

We also check whether the pivot element is already in its final position before swapping it with the element at i+1. If the pivot is already in its final position, the swap is unnecessary and we skip it.

Note that the overall order of the quicksort algorithm is not changed by this modification, but the number of swaps performed during the partition phase is reduced, which can result in faster execution times for large arrays.