**Part B:**

**A:** Write a java function to calculate the salary of an employee.

package Assignment3;  
  
import java.util.\*;  
  
public class SalaryCalculation {  
  
 // Time Complexity : O(1)  
 // Space Complexity: O(1)  
 public static double employeeSalary(double hours) {  
  
 if (hours > 48) {  
 System.*err*.println(**"Exceeds the maximum hours allowed!"**);  
 System.*exit*(1);  
 }  
  
 if (hours <= 36)  
 return hours \* 15;  
 else if (hours <= 36 + 5){  
  
 double salary = 36 \* 15 + (hours - 36) \* 15 \* 1.5;  
 return salary;  
 } else {  
  
 double salary = 36 \* 15 + 5 \* 15 \* 1.5 + (hours - 36 - 5) \* 15 \* 2;  
 return salary;  
 }  
 // None extra time used. The time complexity is O(1).  
 // None extra space used. The space complexity is O(1).  
 }  
  
 public static double getDouble(Scanner scanner) {  
 while (!scanner.hasNextDouble()) {  
  
 System.*out*.println(**"Invalid input, please enter a number."**);  
 scanner.next();  
 }  
 return scanner.nextDouble();  
 }  
  
 public static void main(String[] args){  
  
 Scanner inputHours = new Scanner(System.*in*);  
 double hoursWorked;  
  
 System.*out*.println(**"Enter number of hours the employee worked:"**);  
 hoursWorked = *getDouble*(inputHours);  
  
 System.*out*.println(**"The salary is: "** + *employeeSalary*(hoursWorked));  
 }  
}

**B:** Find all unique triplets in the array which gives the sum of zero.

package Assignment3;  
  
import java.util.\*;  
  
public class UniqueTriplets {  
  
 // Time Complexity : O(n^3)  
 // Space Complexity: O(1)  
 public void findTriplets1(int[] arr) {  
  
 // Sorting takes O(nlog(n)) time. The time complexity here is O(nlog(n)).  
 Arrays.*sort*(arr);  
 boolean found = false;  
 // An n loop is used, index i will iterate n times. The time complexity created is O(n).  
 for (int i = 0; i < arr.length - 2; i++) {  
  
 if (arr[i] > 0)  
 break;  
  
 if (i >= 1 && arr[i] == arr[i - 1])  
 continue;  
  
 // An n loop is used, index j will iterate n times. The time complexity here is O(n^2).  
 for (int j = i + 1; j < arr.length - 1; j++) {  
  
 if (j >= i + 2 && arr[j] == arr[j - 1])  
 continue;  
  
 // An n loop is used, index k will iterate n times. The time complexity here is O(n^3).  
 for (int k = arr.length - 1; k > j; k--) {  
  
 if (k <= arr.length - 2 && arr[k] == arr[k + 1])  
 continue;  
  
 if (arr[i] + arr[j] + arr[k] == 0) {  
  
 // None extra space used. The space complexity is O(1).  
 System.*out*.println(**"["** + arr[i] + **","** + arr[j] + **","** + arr[k] + **"]"**);  
 found = true;  
 }  
 }  
 }  
 }  
 if (!found || arr.length < 3)  
 System.*out*.println(**"Triplets do not exist!"**);  
 }  
  
 // Time Complexity : O(n^2)  
 // Space Complexity: O(n)  
 public void findTriplets2(int[] arr) {  
  
 // Sorting takes O(nlog(n)) time. The time complexity here is O(nlog(n)).  
 Arrays.*sort*(arr);  
 HashMap map = new HashMap();  
 boolean found = false;  
  
 // An n loop is used, index i will iterate n times. The time complexity created is O(n).  
 for (int i = 0; i < arr.length; i++) {  
  
 // Since we used additional n storage to store key-value in the hash map,  
 // the space complexity here is O(n).  
 map.put(arr[i],i);  
 }  
  
 // An n loop is used, index i will iterate n times. The time complexity created is O(n).  
 for (int i = 0; i < arr.length; i++) {  
  
 if (arr[i] > 0)  
 break;  
  
 if (i >= 1 && arr[i] == arr[i - 1])  
 continue;  
  
 // An n loop is used, index j will iterate n times. The time complexity here is O(n^2).  
 for (int j = i + 1; j < arr.length; j++) {  
  
 if (j >= i + 2 && arr[j] == arr[j - 1])  
 continue;  
  
 int num = - arr[i] - arr[j];  
  
 if (map.containsKey(num) && (int) map.get(num) > j) {  
  
 System.*out*.println(**"["** + arr[i] + **","** + arr[j] + **","** + num + **"]"**);  
 found = true;  
 }  
 }  
 }  
  
 if (!found || arr.length < 3)  
 System.*out*.println(**"Triplets do not exist!"**);  
 }  
  
 // Time Complexity : O(n^2)  
 // Space Complexity: O(1)  
 public void findTriplets3(int[] arr) {  
  
 // Sorting takes O(nlog(n)) time. The time complexity here is O(nlog(n)).  
 Arrays.*sort*(arr);  
 boolean found = false;  
  
 // An n loop is used, index i will iterate n times. The time complexity created is O(n).  
 for (int i = 0; i < arr.length - 2; i++) {  
  
 if (arr[i] > 0)  
 break;  
  
 if (i >= 1 && arr[i] == arr[i - 1])  
 continue;  
  
 int j = i + 1;  
 int k = arr.length - 1;  
  
 // Here we use two pointers. The time complexity here changes to O(n^2).  
 while (j < k) {  
  
 if (arr[i] + arr[j] + arr[k] == 0) {  
  
 // None extra space used. The space complexity is O(1).  
 System.*out*.println(**"["** + arr[i] + **","** + arr[j] + **","** + arr[k] + **"]"**);  
 found = true;  
 j++;  
 k--;  
 while (j < k && arr[j] == arr[j - 1])  
 j++;  
 while (j < k && arr[k] == arr[k + 1])  
 k--;  
 } else if (arr[i] + arr[j] + arr[k] < 0)  
 j++;  
 else  
 k--;  
 }  
 }  
 if (!found || arr.length < 3)  
 System.*out*.println(**"Triplets do not exist!"**);  
 }  
  
 public static void main(String[] args) {  
  
 UniqueTriplets triplets = new UniqueTriplets();  
 int[] arr = {-1, 0, 1, 2, -1, -1, -1, 0, 2, 1, -4, -2, 6, 0, 2};  
// int[] arr= {-1, 0, 1, 2, -1, -4};  
  
// triplets.findTriplets1(arr);  
// triplets.findTriplets2(arr);  
 triplets.findTriplets3(arr);  
 }  
}

**C:** Find the length of the longest substring without repeating characters.

package Assignment3;  
  
import java.util.\*;  
  
public class LongestSubstring {  
  
 // Time Complexity : O(n)  
 // Space Complexity: O(min(m,n))  
 public int lengthOfLongestSubstring(String s) {  
  
 int length = 0;  
 // The space required is upper bounded by the size of the charset m and the size of the string n,  
 // so the space complexity here is O(min(m,n)).  
 HashMap map = new HashMap();  
  
 // An n loop is used, index j will iterate n times. The time complexity here is O(n).  
 for(int i = 0, j = 0; j < s.length(); j++) {  
  
 if (map.containsKey(s.charAt(j))) {  
  
 i = Math.*max*((int)map.get(s.charAt(j)) + 1,i);  
 }  
 map.put(s.charAt(j),j);  
 length = Math.*max*(length, j - i + 1);  
 }  
 return length;  
 }  
  
 public static void main(String[] args) {  
  
 LongestSubstring longestSubstring = new LongestSubstring();  
 String s = **"pwwkew"**;  
// String s = "abcabcbb";  
// String s = "bbbbb";  
 int length = longestSubstring.lengthOfLongestSubstring(s);  
  
 System.*out*.println(**"Given "** + **"'"** + s + **"'"** + **", the length of the longest substring is "** + length);  
 }  
}