Assessment of Intech Additive Solution

a) String compression:

```
public class RLE {
    public static String compress(String input) {
        StringBuilder output = new StringBuilder();
        int count = 1;
        char prev = input.charAt(0);

        for (int i = 1; i < input.length(); i++) {
            char current = input.charAt(i);
            if (current == prev) {
                count++;
            } else {
                output.append(prev);
                output.append(count);
                prev = current;
                count = 1;
            }
        }

        // Append the last run
        output.append(prev);
        output.append(count);

        // Compress further
        return compress2(output.toString());
    }

    public static String decompress(String input) {
        // Decompress first
        String decompressed = decompress2(input);

        StringBuilder output = new StringBuilder();
        int count = 0;

        for (int i = 0; i < decompressed.length(); i++) {
            char current = decompressed.charAt(i);
            if (Character.isDigit(current)) {
                count = count * 10 + Character.getNumericValue(current);
            } else {
                for (int j = 0; j < count; j++) {
                    output.append(current);
                }
                count = 0;
            }
        }
```

```
    return output.toString();
}

public static String compress2(String input) {
    StringBuilder output = new StringBuilder();
    int count = 1;
    char prev = input.charAt(0);

    for (int i = 1; i < input.length(); i++) {
        char current = input.charAt(i);
        if (current == prev) {
            count++;
        } else {
            if (count == 2) {
                output.append(prev);
            } else if (count > 2) {
                output.append(count);
                output.append(prev);
            }
            prev = current;
            count = 1;
        }
    }

    // Append the last run
    if (count == 2) {
        output.append(prev);
    } else if (count > 2) {
        output.append(count);
        output.append(prev);
    }

    return output.toString();
}

public static String decompress2(String input) {
    StringBuilder output = new StringBuilder();
    int count = 0;

    for (int i = 0; i < input.length(); i++) {
        char current = input.charAt(i);
        if (Character.isDigit(current)) {
            count = count * 10 + Character.getNumericValue(current);
        } else {
            if (count == 0) {
                output.append(current);
            } else if (count == 1) {
                output.append(current);
                count = 0;
            } else if (count > 1) {
                for (int j = 0; j < count - 1; j++) {
```

```
                output.append(current);
            }
            count = 0;
        }
    }
}

return output.toString();
}

public static void main(String[] args) {
    String input = "aabbcaaaccc";
    String compressed = compress(input);
    String decompressed = decompress(compressed);

    System.out.println("Input: " + input);
    System.out.println("Compressed: " + compressed);
    System.out.println("Decompressed: " + decompressed);
```

To improve memory usage, one approach is to use compression algorithms to reduce the size of the data being stored. One such algorithm is run-length encoding (RLE), which can be used to compress repetitive data by replacing runs of identical values with a count of the number of repetitions and the value itself.

Here's an implementation of RLE compression in Java that can handle the bonus requirements:

B) Linked List:

One way to find the kth to the last element of a linked list without knowing the length of the list is to use a two-pointer approach. We can initialize two pointers, p1 and p2, both pointing to the head of the linked list. We can then move p2 k positions forward in the linked list. After that, we can move both p1 and p2 one position forward until p2 reaches the end of the linked list. At this point, p1 will be pointing to the kth to the last element of the linked list.

```
public ListNode findKthToLast(ListNode head, int k) {
   ListNode p1 = head;
   ListNode p2 = head;

   // Move p2 k positions forward
   for (int i = 0; i < k; i++) {
      if (p2 == null) {
         return null; // Error: k is greater than the length of the list
      }
      p2 = p2.next;
   }

   // Move both pointers one position forward until p2 reaches the end of the list
   while (p2 != null) {
      p1 = p1.next;
      p2 = p2.next;
   }

   return p1;
}
```

In this implementation, we assume that the linked list is represented using the ListNode class, where each node has a next pointer to the next node in the list. The method takes as input the head of the linked list and the value of k. If k is greater than the length of the list, the method returns null. Otherwise, it returns the kth to the last element of the linked list.

C) Stack minimum:

```
        public class MinStack {
    private Stack<Integer> stack = new Stack<>();
    private Stack<Integer> minStack = new Stack<>();
    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek()) {
            minStack.push(x);
        }
    }
    public void pop() {
        if (stack.isEmpty()) {
            return;
        }
        int x = stack.pop();
        if (x == minStack.peek()) {
            minStack.pop();
        }
    }
    public int top() {
        if (stack.isEmpty()) {
            return -1;
        }
        return stack.peek();
    }
    public int getMin() {
        if (minStack.isEmpty()) {
            return -1;
        }
        return minStack.peek();
    }

    public static void main(String[] args) {
```

```
        MinStack stack = new MinStack();
        stack.push(3);
        stack.push(1);
        stack.push(2);
        stack.push(5);
        stack.push(0);

        System.out.println("Minimum element: " + stack.getMin()); // 0
        stack.pop();
        stack.pop();
        System.out.println("Minimum element: " + stack.getMin()); // 1
        stack.push(-1);
        System.out.println("Minimum element: " + stack.getMin()); // -1
    }
}
```

To implement a Stack with a min function that returns the minimum element in the stack in constant time (O(1)), we can use two stacks: a main stack to store the elements and an auxiliary stack to store the minimums.

Each time an element is pushed onto the main stack, we compare it with the top element of the auxiliary stack. If it is smaller or equal, we push it onto the auxiliary stack as well. When an element is popped from the main stack, we check if it is the same as the top element of the auxiliary stack. If it is, we also pop it from the auxiliary stack. The min function simply returns the top element of the auxiliary stack.

Here's an implementation of the Stack with a min function in Java:

Bonus 1: A real-world use case where a stack is a better data structure than an array is in the implementation of a web browser's back button functionality. When the user visits a website, the browser can push the URL onto a stack. When the user clicks the back button, the browser can simply pop the top URL from the stack to return to the previous page. This approach is more efficient than storing all visited URLs in an array and iterating through it to go back to a previous page. Additionally, the stack approach takes up less memory than an array since it only stores the most recent URLs.

D)

```
public static int trapping_rain_water(int[] A, int N)
{
   int res = 0;
   for(int i = 0; i < N ; i++)
   {
      int left_max= arr[i];
      for(int j = i - 1; j >= 0; j--)
      {
         left_max = Math.max(left_max, A[j]);
      }
      int right_max = A[i];
      for(int j = i + 1; j < n; j++)
      {
         right_max = Math.max(right_max, A[j]);
      }
      res += Math.min(left_max, right_max) - A[i];
   }
   return res;
}
```

---

e)

Given a list of coin denominations and an amount of change to be tendered, the problem is to find the minimum number of coins required to make the exact change. This is known as the coin change problem.

A greedy algorithm for the coin change problem involves selecting the largest denomination coin that can be used to make the change and subtracting it from the remaining amount until the amount becomes zero or negative. This process is repeated until the amount becomes zero.

However, the greedy algorithm may not always result in the minimum number of coins. For example, if the coin denominations are {1, 5, 10} and the amount of change to be tendered is 15, the greedy algorithm would select 10, 5 and 1, requiring a total of 3 coins. However, the optimal solution would be to use 3 coins of denomination 5, requiring a total of 3 coins.

Dynamic programming can be used to solve the coin change problem optimally. The idea is to use a table to store the minimum number of coins required to make change for each value from 0 to the desired amount. The table is filled in a bottom-up manner, using the minimum number of coins required for smaller values to calculate the minimum number of coins required for larger values. The time complexity of the dynamic programming approach is O(amount * numCoins), where amount is the amount of change to be tendered and numCoins is the number of coin denominations.

For the given input of coin denominations {1, 7, 2, 5} and the change to be tendered of 8 and 10, the optimal solution is to use 2 and 5 coins respectively.

Bonus question:

To find the largest possible number by removing one digit from a given number N, we can iterate through each digit of N and remove it one by one to obtain a new number. The largest possible number is the maximum of all the new numbers obtained. For example, if N is 19374, the largest possible number by removing one digit is 9374.

This solution is part of a greedy algorithm because we are iteratively selecting the maximum digit to remove at each step without considering the impact of the removal on the remaining digits. It is possible that a smaller digit may need to be removed first to obtain the largest possible number. However, since the number of digits is small, the greedy approach works well in practice.

F) :

Dot product and cross product are mathematical operations used in vector algebra.

Dot product: The dot product of two vectors A and B is defined as the product of their magnitudes and the cosine of the angle between them. It is denoted as A . B. The dot product of two vectors is a scalar quantity.

Cross product: The cross product of two vectors A and B is defined as a vector that is perpendicular to both A and B. It is denoted as A x B. The cross product of two vectors is a vector quantity.

In graphics environment, dot product is used to calculate the angle between two vectors, which is useful in lighting calculations such as diffuse and specular lighting. Dot product can also be used to project one vector onto another.

Cross product is used in graphics environment for a variety of purposes such as calculating surface normals, determining the orientation of triangles, and computing the rotation axis between two vectors. It is also used in physics simulations to calculate torque and angular momentum.

Here are some resources for further reading:

Dot product: https://en.wikipedia.org/wiki/Dot_product

Cross product: https://en.wikipedia.org/wiki/Cross_product

Use of dot product in graphics: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/dot-product-usage-in-3d-graphics

Use of cross product in graphics: https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/cross-product-and-handiness

Bonus answer:

Intersection between a ray and a plane:

To calculate the intersection between a ray and a plane, we first need to determine if the ray is intersecting the plane. This can be done by calculating the dot product of the ray direction and the plane normal. If the dot product is zero, the ray is parallel to the plane and does not intersect it. If the dot product is non-zero, the ray intersects the plane at a point P = O + tD, where O is the origin of the ray, D is the direction of the ray, and t is the distance along the ray at which the intersection occurs.

Intersection between a ray and a sphere:

To calculate the intersection between a ray and a sphere, we first need to determine if the ray is intersecting the sphere. This can be done by calculating the discriminant of the quadratic equation that represents the intersection between the ray and the sphere. If the discriminant is negative, the ray does not intersect the

sphere. If the discriminant is zero, the ray intersects the sphere at a single point. If the discriminant is positive, the ray intersects the sphere at two points.

Intersection between a ray and a triangle:

To calculate the intersection between a ray and a triangle, we first need to determine if the ray is intersecting the plane of the triangle. This can be done by calculating the dot product of the ray direction and the triangle normal. If the dot product is zero, the ray is parallel to the triangle and does not intersect it. If the dot product is non-zero, the ray intersects the plane of the triangle. We then need to determine if the intersection point is inside the triangle by checking if it lies within the bounds of the triangle edges.

Here are some resources for further reading:

Ray-plane intersection: https://en.wikipedia.org/wiki/Line%E2%80%93plane_intersection

Ray-sphere intersection: https://en.wikipedia.org/wiki/Line%E2%80%93sphere_intersection

Ray-triangle intersection:
https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm

---

G):

As an AI language model, I have not written any code myself. However, I can tell you about one interesting engineering subject - robotics!

Robotics is a field of engineering and science that involves the design, construction, operation, and use of robots. Robots are used in a variety of applications, including manufacturing, transportation, healthcare, space exploration, and more.

One particular aspect of robotics that is fascinating is the development of autonomous robots that can operate independently without human intervention. This requires the integration of various technologies, such as computer vision, machine learning, sensors, and actuators.

One recent example of an impressive application of robotics is the development of autonomous drones for package delivery. Companies like Amazon and UPS are currently testing drone delivery systems that can fly packages directly to customers' doorsteps, without the need for a human pilot.

The drones are equipped with a variety of sensors, such as cameras and lidar, which allow them to navigate through obstacles and avoid collisions. They also use machine learning algorithms to analyze the environment and make decisions about the best path to take.

Overall, robotics is an exciting and rapidly evolving field that has the potential to transform many industries and improve people's lives in countless ways.

H):

If a source code is randomly crashing and not always in the same place, there could be several reasons for it. Some possible reasons are:

Memory issues: The code may be trying to access memory that it shouldn't be accessing, leading to segmentation faults or other memory-related errors.

Concurrency issues: The code may be using multiple threads or processes, and there may be race conditions or other synchronization issues that are causing crashes.

Input data issues: The code may be receiving input data that is causing unexpected behavior or triggering edge cases that the code is not prepared to handle.

System/environmental issues: The code may be sensitive to changes in the environment, such as changes in system resources, network connectivity, or other external factors.

To isolate the cause of the random crashes, I would take the following steps:

Collect crash information: Use a debugger or other monitoring tool to collect information about when and where the crashes are occurring, as well as any error messages or other diagnostic information that is available.

Analyze the code: Review the code to identify any potential issues that could be causing the crashes, such as memory access violations or synchronization problems.

Test different scenarios: Create test cases that exercise different parts of the code and different inputs to try to reproduce the crashes in a controlled environment.

Monitor system resources: Use system monitoring tools to check for any unusual spikes or drops in system resources, such as CPU usage, memory usage, or disk I/O.

Consider external factors: Review any changes that have been made to the system or environment, such as software updates or changes to network connectivity, that may be affecting the code.

In terms of computer architecture, the underlying hardware and system architecture can also have an impact on the code's behavior and stability. For example, issues such as cache misses, memory alignment, and hardware errors can all cause crashes or unexpected behavior. Understanding the details of the underlying architecture can help in identifying and diagnosing these types of issues.