a ) String compression

Implement a method to perform string compression. E.g. ‘aabcccccaaa’ should be a2b1c5a3. The code to implement this is given in the link - <https://www.educative.io/answers/string-compression-using-run-length-encoding>

Think about memory occupied and how it can be improved.

Bonus 1:

The answer should be taken into second compressor and compress further.

E.g. a2b2c1a3c3 should become ab2c1ac3

Bonus 2: decompress2

ab2c1ac3 should return aabbcaaaccc.

Think about how you will test this code.

Ans:-> code:

#include <iostream>

#include <sstream>

#include <vector>

// Function to perform basic string compression

std::string stringCompression(const std::string &s)

{

    std::string compressed;

    int count = 1;

    // Iterate through the characters of the string

    for (int i = 1; i < s.length(); ++i)

    {

        // If the current character is the same as the previous one, increment count

        if (s[i] == s[i - 1])

        {

            count++;

        }

        else

        {

            // Append the character and its count to the compressed string

            compressed += s[i - 1] + (count > 1 ? std::to\_string(count) : "");

            count = 1;

        }

    }

    // Append the last character and its count to the compressed string

    compressed += s.back() + (count > 1 ? std::to\_string(count) : "");

    // Return the compressed string if it is shorter, otherwise return the original string

    return compressed.length() < s.length() ? compressed : s;

}

// Bonus compressor that handles single counts differently

std::string bonusCompressor(const std::string &s)

{

    std::string compressed;

    int count = 1;

    // Iterate through the characters of the string

    for (int i = 1; i < s.length(); ++i)

    {

        // If the current character is the same as the previous one, increment count

        if (s[i] == s[i - 1])

        {

            count++;

        }

        else

        {

            // Append the character and its count to the compressed string

            compressed += s[i - 1] + (count > 1 ? std::to\_string(count) : "");

            count = 1;

        }

    }

    // Append the last character and its count to the compressed string

    compressed += s.back() + (count > 1 ? std::to\_string(count) : "");

    // Return the compressed string

    return compressed;

}

// Decompressor to reconstruct the original string from the compressed version

std::string decompress(const std::string &s)

{

    std::string decompressed;

    int i = 0;

    // Iterate through the characters of the compressed string

    while (i < s.length())

    {

        // Get the current character

        char currentChar = s[i++];

        std::string countStr;

        // Extract the count of the current character

        while (i < s.length() && isdigit(s[i]))

        {

            countStr += s[i++];

        }

        // Convert the count from string to integer (default to 1 if countStr is empty)

        int count = countStr.empty() ? 1 : std::stoi(countStr);

        // Append the reconstructed characters to the decompressed string

        decompressed += std::string(count, currentChar);

    }

    // Return the decompressed string

    return decompressed;

}

int main()

{

    // Test the code with an example string

    std::string originalStr = "aabcccccaaa";

    // Perform basic string compression

    std::string compressedStr = stringCompression(originalStr);

    std::cout << "Compressed String: " << compressedStr << std::endl;

    // Use the bonus compressor

    std::string bonusCompressedStr = bonusCompressor(compressedStr);

    std::cout << "Bonus Compressed String: " << bonusCompressedStr << std::endl;

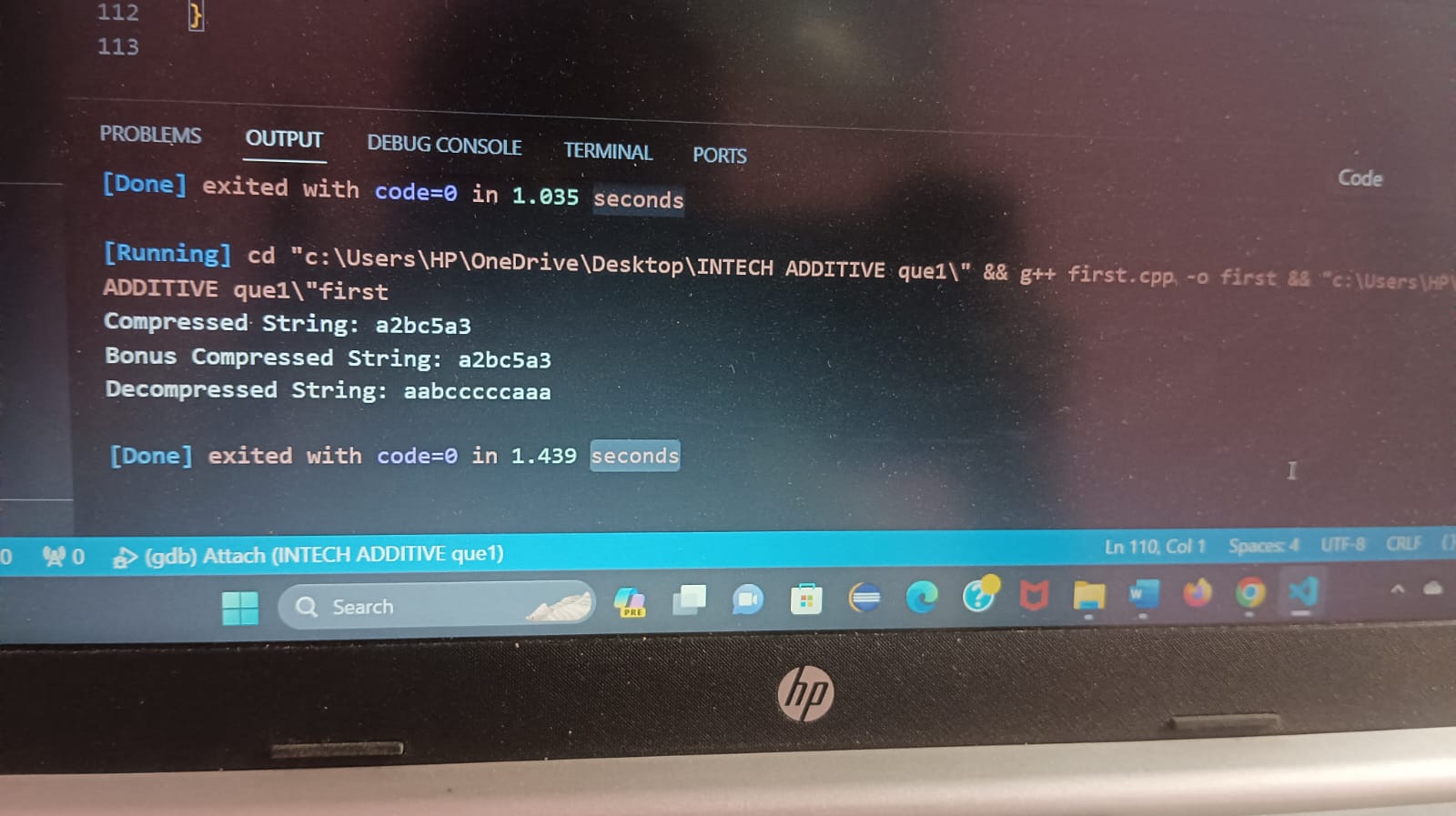
    // Decompress the bonus compressed string

    std::string decompressedStr = decompress(bonusCompressedStr);

    std::cout << "Decompressed String: " << decompressedStr << std::endl;

    return 0;

}

Output:

Git hub link:

b) Linked List - The link shows a program to find the nth element of a linked list. <https://www.geeksforgeeks.org/nth-node-from-the-end-of-a-linked-list/>

Find a way to find the kth to the last element of linked list ( assume length of linked list is not known)

Bonus 1:

Can you minimize the number of times you run through the loop.

Code:

#include <iostream>

struct Node

{

    int data;

    Node \*next;

};

Node \*findKthToLast(Node \*head, int k)

{

    Node \*fast = head;

    Node \*slow = head;

    // Move the fast pointer k steps ahead

    for (int i = 0; i < k; i++)

    {

        if (fast == nullptr)

        {

            return nullptr; // k is greater than the length of the linked list

        }

        fast = fast->next;

    }

    // Move both pointers until the fast pointer reaches the end

    while (fast != nullptr)

    {

        fast = fast->next;

        slow = slow->next;

    }

    return slow; // slow pointer is now pointing to the kth to the last element

}

int main()

{

    // Create a sample linked list

    Node \*head = new Node{1, nullptr};

    Node \*second = new Node{2, nullptr};

    Node \*third = new Node{3, nullptr};

    Node \*fourth = new Node{4, nullptr};

    Node \*fifth = new Node{5, nullptr};

    head->next = second;

    second->next = third;

    third->next = fourth;

    fourth->next = fifth;

    int k = 2; // Find the 2nd to the last element

    Node \*kthToLast = findKthToLast(head, k);

    if (kthToLast != nullptr)

    {

        std::cout << "The " << k << "th to the last element is: " << kthToLast->data << std::endl;

    }

    else

    {

        std::cout << "Invalid input or k is greater than the length of the linked list." << std::endl;

    }

    // Clean up memory

    delete head;

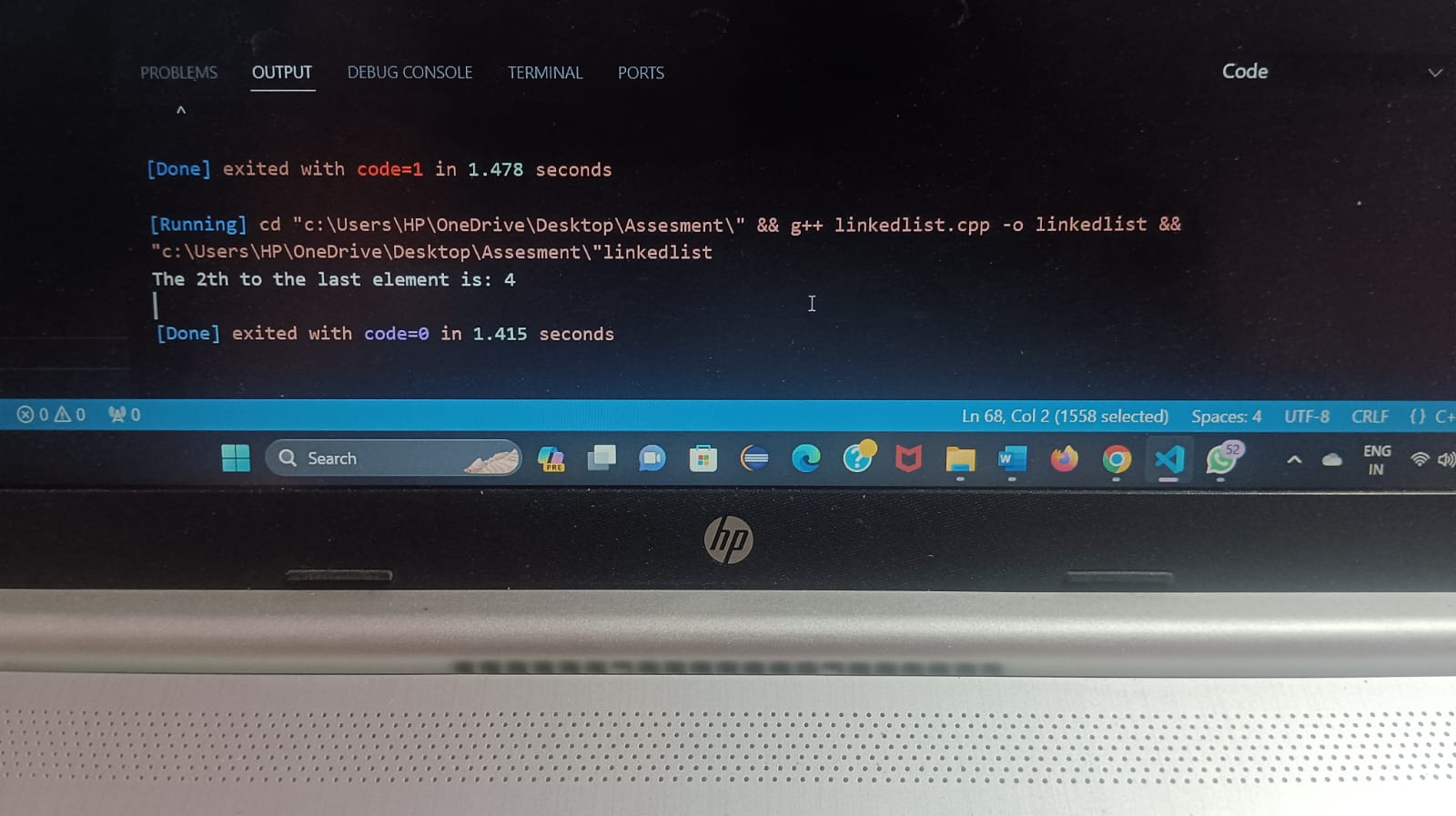
    delete second;

    delete third;

    delete fourth;

    delete fifth;

    return 0; }

output:

c) Stack minimum- Details of stack data structure is available in <https://www.geeksforgeeks.org/stack-data-structure/>

Stack has functions of push and pop. Can you also add a function ‘min’ to the stack and it should also execute in O(1).

If you are not aware of O(1), refer to some videos online. E.g. <https://en.wikipedia.org/wiki/Big_O_notation>

Bonus 1 –

Explain one real world use case where stack is better used data structure than arrays.

Code:

#include <iostream>

#include <stack>

class MinStack {

private:

    std::stack<int> mainStack;

    std::stack<int> minStack;

public:

    void push(int value) {

        mainStack.push(value);

        if (minStack.empty() || value <= minStack.top()) {

            minStack.push(value);

        }

    }

    void pop() {

        if (mainStack.empty()) {

            return;

        }

        if (mainStack.top() == minStack.top()) {

            minStack.pop();

        }

        mainStack.pop();

    }

    int min() {

        if (minStack.empty()) {

            return -1; // Or any other appropriate value or error handling

        }

        return minStack.top();

    }

};

int main() {

    MinStack stack;

    stack.push(5);

    stack.push(2);

    stack.push(7);

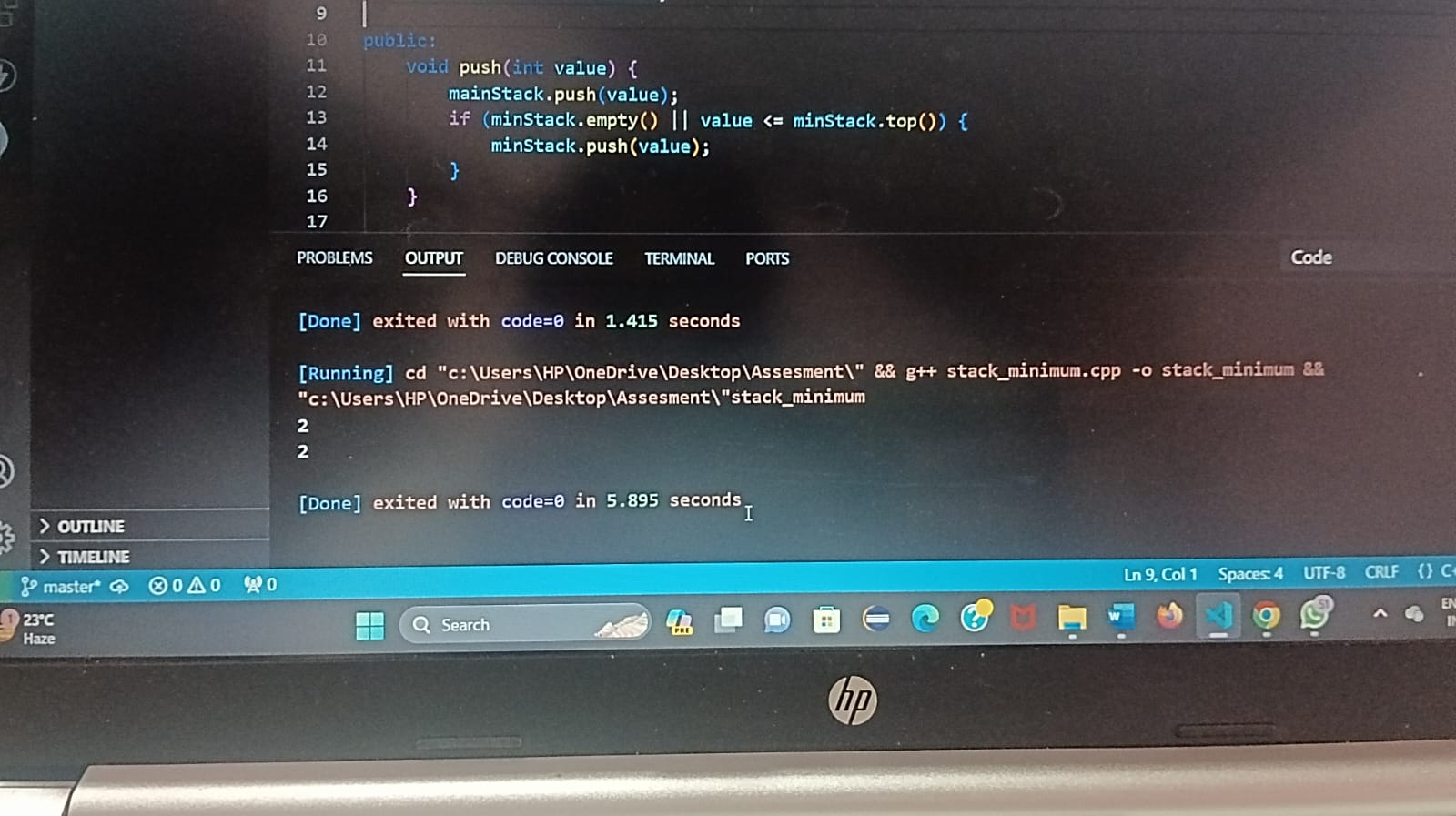
    std::cout << stack.min() << std::endl;  // Output: 2

    stack.pop();

    std::cout << stack.min() << std::endl;  // Output: 2

    return 0;

}

Output:

Explain one real world use case where stack is better used data structure than arrays:-

real-world use case where a stack is a better choice than an array is in the implementation of the back button in web browsers.

When you browse the internet, you often navigate through different web pages by clicking on links. The back button allows you to go back to the previously visited page. A stack data structure is commonly used to keep track of the visited pages.

Each time you visit a new page, you can push the URL or page information onto the stack. When you click the back button, the most recent page is popped from the stack, allowing you to return to the previous page. This behaviour follows the LIFO principle of a stack.

Using an array for this functionality would require shifting elements whenever you go back, which can be inefficient. With a stack, you can easily navigate through the visited pages by popping the previous page from the stack.

So, in the case of web browsers, a stack is a better choice than an array because it provides an efficient way to implement the back button functionality and maintain the browsing history.

1. Given an array of integers representing the elevation of a roof structure at  
   various positions, each position is separated by a unit length, Write a program  
   to determine the amount of water that will be trapped on the roof after heavy  
   rainfall.

Example:  
input : [2 1 3 0 1 2 3]

Ans : 7 units of water will be trapped

Code: Here's the C++ version of the program to determine the amount of water that will be trapped on the roof after heavy rainfall:

#include <iostream>

#include <vector>

#include <algorithm>

int calculateWaterTrapped(const std::vector<int>& heights) {

int n = heights.size();

if (n < 3) {

return 0;

}

std::vector<int> leftMax(n, 0);

std::vector<int> rightMax(n, 0);

leftMax[0] = heights[0];

for (int i = 1; i < n; i++) {

leftMax[i] = std::max(leftMax[i-1], heights[i]);

}

rightMax[n-1] = heights[n-1];

for (int i = n-2; i >= 0; i--) {

rightMax[i] = std::max(rightMax[i+1], heights[i]);

}

int waterTrapped = 0;

for (int i = 0; i < n; i++) {

waterTrapped += std::min(leftMax[i], rightMax[i]) - heights[i];

}

return waterTrapped;

}

int main() {

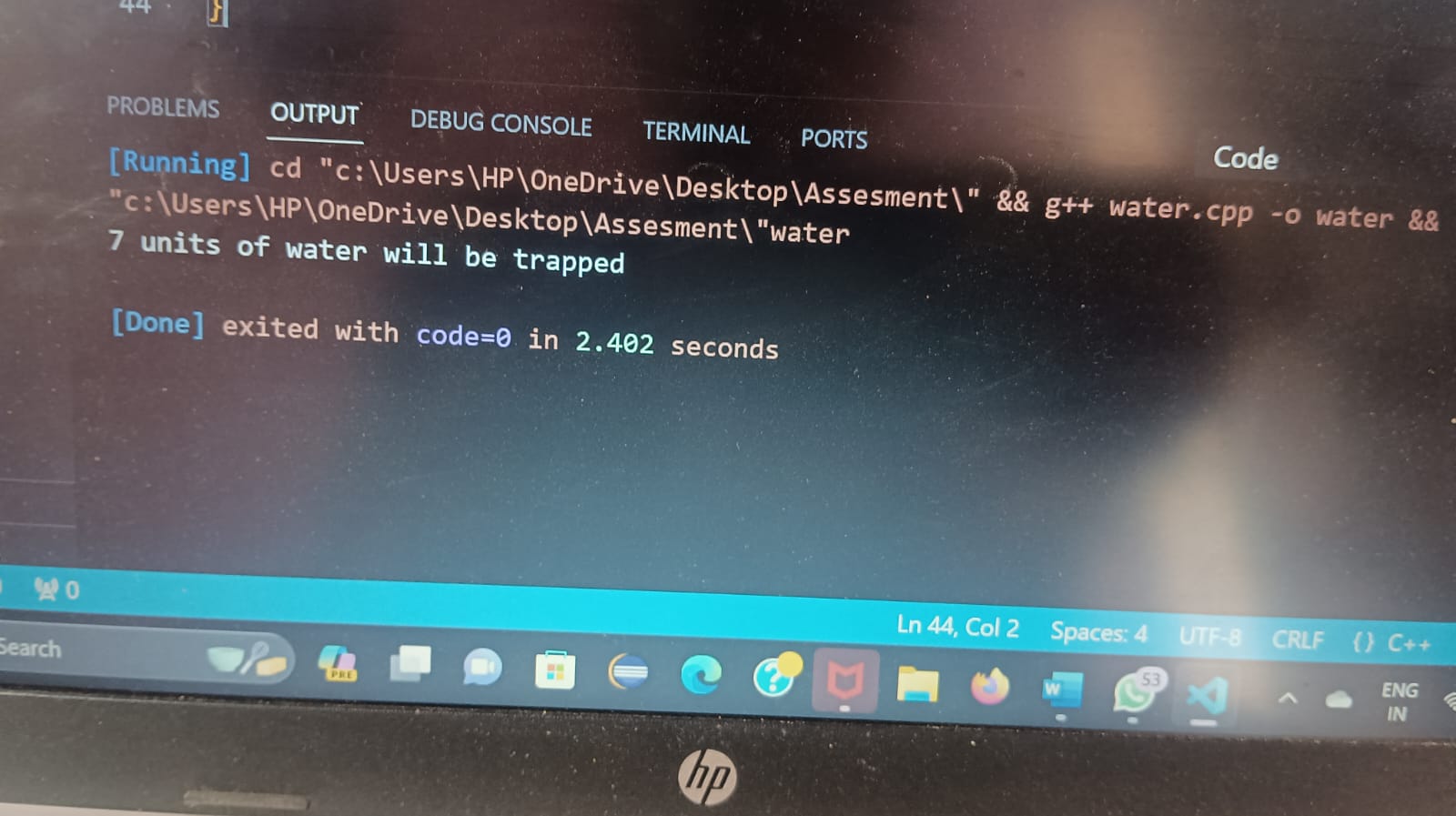
std::vector<int> heights = {2, 1, 3, 0, 1, 2, 3};

int water = calculateWaterTrapped(heights);

std::cout << water << " units of water will be trapped" << std::endl;

return 0;

}

Output:

e)You will be given a list coin denominations that you can use to tender change to  
your customers, find the most optimum way to tender the exact change to your  
customers , the optimum is when you use the least number of coins.

Example:  
input => [ 1, 2, 5, 8, 10] (available coins)  
Input => 7 (Change to be given)  
Ans : [ 2, 5 ]

Explain all the scenarios in better words and simpler to understand format compared to explanation available in the link below:

<https://www.geeksforgeeks.org/coin-change-dp-7/>

Ans):> Sure, let me explain the scenario in a simpler way. Imagine you're a cashier and you have a list of available coins that you can use to give change to your customers. The goal is to find the best way to give the exact change using the fewest number of coins.

For example, let's say you have the following available coins: 1, 2, 5, 8, and 10. And a customer needs 7 units of change. We need to figure out the best combination of coins to make up that amount.

In this case, the most optimum way to give the exact change of 7 units is by using a combination of 2 and 5 coins. By using one 2-unit coin and one 5-unit coin, we can make up the total of 7 units. This way, we are using the least number of coins possible.

The idea is to find the largest coin denomination that is less than or equal to the remaining change, and repeat this process until the change becomes zero. By choosing the largest coin denomination each time, we can minimize the number of coins used.

Que) Explain what is a greedy algorithm and how dynamic programming helps in this case.

Ans:> A greedy algorithm is a simple and intuitive approach to problem-solving. It involves making the locally optimal choice at each step with the hope that it will lead to a globally optimal solution. In other words, a greedy algorithm makes the best choice at each step without considering the overall consequences.

On the other hand, dynamic programming is a technique used to solve complex problems by breaking them down into smaller overlapping subproblems. It solves each subproblem only once and stores the result, so that when the same subproblem is encountered again, it can be quickly retrieved instead of recomputing it.

Dynamic programming can be used to optimize the solution provided by a greedy algorithm. While a greedy algorithm focuses on making the best choice at each step, it may not always lead to the most optimal solution overall. By using dynamic programming, we can evaluate and store the results of each subproblem, allowing us to make better-informed decisions and potentially find a globally optimal solution.

In some cases, a greedy algorithm alone may provide the optimal solution, but in more complex scenarios, dynamic programming can help improve the efficiency and accuracy of the solution.

Bonus question:

given a number N, remove one digit and print the largest possible number.

E.g.

5 – 1234

2945

9273

3954

19374

Answers:

234

945

973

954

9374

ANS:->

Code: #include <iostream>

#include <string>

using namespace std;

string removeOneDigitAndGetLargest(int N) {

// Convert the number to a string for easy manipulation

string numStr = to\_string(N);

int n = numStr.length();

// Iterate through the digits to find the first digit that is greater than the next one

int i = 0;

while (i < n - 1 && numStr[i] >= numStr[i + 1]) {

i++;

}

// Remove the first decreasing digit

numStr.erase(i, 1);

return numStr;

}

int main() {

int num;

// Input the number

cout << "Enter a number: ";

cin >> num;

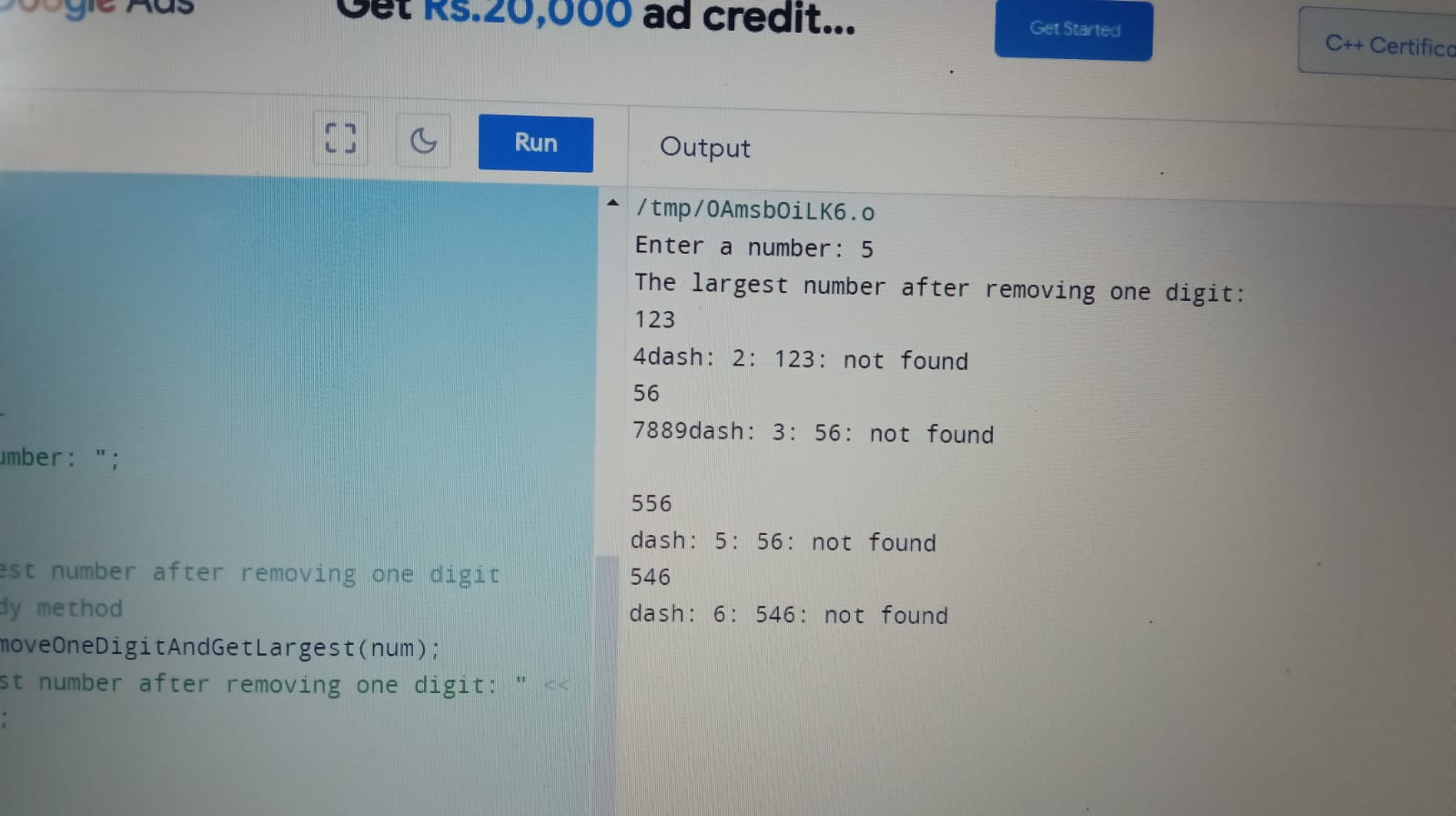
// Output the largest number after removing one digit using the greedy method

string result = removeOneDigitAndGetLargest(num);

cout << "The largest number after removing one digit: " << result << endl;

return 0;

}

Output:

Why is the above solution part of a greedy algorithm?

Ans:> While the problem may seem simple, the application of the greedy approach ensures that at each step, the algorithm makes the locally optimal choice, and these choices collectively lead to an overall optimal solution — in this case, the largest possible number after removing one digit.

1. What is dot product and cross product? Explain use cases of where dot product is used and cross product is used in graphics environment. Add links to places where you studied this information and get back with the understanding.

Dot Product:

In computer graphics, the dot product is used in various applications. For example, in 3D modeling and animation, the dot product is used to calculate the intensity of light hitting a surface. This helps determine how bright or dim a specific point on an object should appear based on the angle between the surface normal and the light source direction.

Cross Product:

The cross product is also widely used in graphics. One example is in calculating surface normals. In computer graphics, surface normals are essential for shading and rendering realistic objects. The cross product helps determine the direction of the surface normal, which affects how light interacts with the object's surface.

Another example of the cross product's usage is in calculating tangent and bitangent vectors for normal mapping. Normal mapping is a technique used to add intricate details to 3D models. The cross product is employed to calculate tangent and bitangent vectors, which are then used to map textures onto the model's surface accurate

Link: https://study.com/academy/lesson/the-dot-product-and-vectors-definition-formula.html

Bonus - How do you calculate the intersection between a ray and a plane/sphere/triangle?

Ans:> To calculate the intersection between a ray and a plane, sphere, or triangle in a graphics environment, different methods are used for each shape.

1. Ray-Plane Intersection:

For a ray and a plane, you can use the parametric equation of a line to find the point of intersection. By substituting the ray's equation into the plane's equation, you can solve for the parameter value that gives the intersection point.

2. Ray-Sphere Intersection:

To find the intersection between a ray and a sphere, you can use the quadratic formula. By substituting the ray's equation into the sphere's equation, you can solve for the parameter values that give the intersection points, if they exist.

3. Ray-Triangle Intersection:

Calculating the intersection between a ray and a triangle involves a more complex algorithm. One commonly used method is the Möller-Trumbore algorithm. It uses the ray's direction, origin, and the triangle's vertices to determine if an intersection occurs and the barycentric coordinates of the intersection point.

These intersection calculations are fundamental in computer graphics for tasks like ray tracing, collision detection, and rendering realistic scenes.

1. Explain a piece of code that you wrote which you are proud of? If you have not written any code, please write your favorite subject in engineering studies. We can go deep into that subject.

Ans:> Certainly! Computer security, also known as cybersecurity, is a critical aspect of protecting computer systems, networks, and data from unauthorized access, attacks, damage, or theft. Here are some valuable pieces of information about computer security:

1. **Confidentiality, Integrity, and Availability (CIA):**
   * The core principles of computer security are often summarized as the CIA triad.
   * **Confidentiality:** Ensures that information is only accessible to those authorized to view it.
   * **Integrity:** Guarantees that information is accurate and unaltered.
   * **Availability:** Ensures that systems and data are available and accessible when needed.
2. **Common Cyber Threats:**
   * **Malware:** Malicious software designed to harm or exploit computers.
   * **Phishing:** Deceptive attempts to obtain sensitive information by pretending to be a trustworthy entity.
   * **Denial of Service (DoS) Attacks:** Overloading a system or network to disrupt its normal functioning.
   * **Man-in-the-Middle Attacks:** Intercepting communication between two parties without their knowledge.
3. **Authentication and Authorization:**
   * **Authentication:** Verifying the identity of users or systems through passwords, biometrics, or multi-factor authentication.
   * **Authorization:** Granting or restricting access to resources based on authenticated user privileges.
4. **Patch Management:**
   * Regularly updating and patching software and operating systems is crucial to address vulnerabilities and prevent exploitation by attackers.
5. **Firewalls and Intrusion Detection Systems (IDS):**
   * Firewalls monitor and control incoming and outgoing network traffic, acting as a barrier between a trusted network and untrusted networks.
   * IDSs identify and respond to suspicious activities or security policy violations.
6. **Encryption:**
   * Encrypting data helps protect it during transmission and storage by converting it into a secure, unreadable format that can only be deciphered with the proper decryption key.
7. **Security Policies and Employee Training:**
   * Establishing and enforcing security policies helps create a secure environment.
   * Ongoing employee training is crucial to raise awareness about security best practices and potential threats.
8. **Incident Response and Recovery:**
   * Having a well-defined incident response plan helps organizations react effectively to security incidents.
   * Regularly backing up data is essential for quick recovery in the event of a security breach.
9. **Mobile Device Security:**
   * Securing mobile devices is vital as they often store sensitive information and may connect to corporate networks.
   * Implementing device encryption, strong authentication, and mobile device management (MDM) solutions enhances security.
10. **Compliance and Legal Considerations:**
    * Many industries have regulatory requirements regarding data protection and security (e.g., GDPR, HIPAA).
    * Adhering to these regulations is not only a legal obligation but also essential for maintaining trust with users and clients.
11. Random crashes – you are given a source code to test and it randomly crashes and it never crashes in the same place ( you have attached a debugger and you find this). Explain what all you would suspect and how would you go about with isolating the cause.

Ans:>

Random crashes in a software application can be challenging to diagnose, but there are several common issues and debugging strategies you can employ to isolate the cause. Here's a systematic approach to help identify and fix the problem:

**1. Memory Issues:**

* **Suspected Causes:**
  + Memory leaks
  + Invalid memory access
  + Dangling pointers
* **Debugging Steps:**
  + Use tools like Valgrind or AddressSanitizer to check for memory leaks and invalid memory access.
  + Verify proper memory deallocation after allocation.

**2. Thread Issues:**

* **Suspected Causes:**
  + Race conditions
  + Deadlocks
  + Incorrect thread synchronization
* **Debugging Steps:**
  + Employ thread sanitizers or tools like Helgrind to detect threading issues.
  + Review thread synchronization mechanisms for correctness.

**3. External Dependencies:**

* **Suspected Causes:**
  + Issues with external libraries or components
  + Incorrect API usage
* **Debugging Steps:**
  + Verify the compatibility of external dependencies.
  + Review the documentation and usage of external APIs.

**4. Input Data Issues:**

* **Suspected Causes:**
  + Unexpected or malformed input data
  + Buffer overflows due to incorrect input handling
* **Debugging Steps:**
  + Validate input data and ensure robust input handling.
  + Use tools like AddressSanitizer to detect buffer overflows.

**5. Hardware or System-Level Issues:**

* **Suspected Causes:**
  + Hardware faults
  + Operating system issues
* **Debugging Steps:**
  + Check system logs for hardware errors.
  + Ensure the application is compatible with the operating system.

**6. Logging and Error Handling:**

* **Suspected Causes:**
  + Inadequate logging or error handling
* **Debugging Steps:**
  + Enhance logging to capture more information during crashes.
  + Review error handling mechanisms for thoroughness.

**7. Dynamic Analysis Tools:**

* **Suspected Causes:**
  + Uninitialized variables
  + Undefined behavior
* **Debugging Steps:**
  + Use dynamic analysis tools like Clang Static Analyzer or UndefinedBehaviorSanitizer.

**8. Version Control:**

* **Suspected Causes:**
  + Recent changes in the codebase
* **Debugging Steps:**
  + Check the version control system for recent code changes.
  + Identify if a recent change might have introduced the issue.

**9. Reproducibility:**

* **Suspected Causes:**
  + Non-deterministic behavior
* **Debugging Steps:**
  + Attempt to make the issue reproducible. This might involve creating specific test cases or scenarios that trigger the crash.

**10. Collaboration:**

* **Suspected Causes:**
  + Lack of expertise
* **Debugging Steps:**
  + Collaborate with team members or seek external expertise.
  + Discuss the issue with developers familiar with the codebase.

Bonus – The deeper you go into computer architecture and explain, better.

Ans:> **1. Memory Issues:**

* **Memory Leaks:**
  + Examine the memory allocation and deallocation patterns.
  + Use tools like Valgrind to identify leaked memory blocks.
  + Investigate if certain code paths fail to release allocated memory.
* **Invalid Memory Access:**
  + Analyze memory access patterns to detect out-of-bounds accesses.
  + Employ tools like AddressSanitizer to identify memory-related issues.
  + Check for pointer arithmetic errors or accessing freed memory.

**2. Concurrency and Parallelism:**

* **Race Conditions:**
  + Inspect code sections involving shared resources for potential race conditions.
  + Use thread sanitizers or advanced tools like ThreadSanitizer to detect race conditions.
  + Consider thread-safe programming paradigms and synchronization mechanisms.
* **Deadlocks:**
  + Review code for instances where multiple locks are acquired in different orders.
  + Analyze thread dumps or utilize deadlock detection tools.
  + Ensure proper lock acquisition and release sequences.

**3. Processor Architecture:**

* **Instruction Set and Architecture-Specific Issues:**
  + Verify that the application is compatible with the target processor architecture.
  + Consider alignment requirements and endianness issues.
  + Review compiler flags and options for optimization and compatibility.

**4. Input Data Handling:**

* **Buffer Overflows:**
  + Scrutinize input parsing and handling code for potential buffer overflows.
  + Utilize AddressSanitizer to detect and pinpoint buffer overflow locations.
  + Implement secure coding practices, such as bounds checking.
* **Data Alignment:**
  + Ensure that data structures are properly aligned in memory.
  + Misaligned access on certain architectures can result in crashes.
  + Check compiler settings to enforce proper alignment.

**5. Operating System Interaction:**

* **System Calls and Interrupts:**
  + Investigate interactions with the operating system, especially during system calls and interrupts.
  + Ensure proper error handling for system calls.
  + Review code sections dealing with signal handling.
* **File System and I/O Operations:**
  + Analyze file I/O operations for potential issues.
  + Check for proper error handling during file operations.
  + Verify compatibility with the file system and device drivers.

**6. Dynamic Linking and Loading:**

* **Shared Libraries and Dynamic Loading:**
  + Examine dependencies on shared libraries.
  + Verify that the correct versions of libraries are being loaded.
  + Investigate issues related to dynamic linking and symbol resolution.

**7. Hardware-Specific Considerations:**

* **CPU Cache and Memory Hierarchy:**
  + Consider how the application utilizes the CPU cache and memory hierarchy.
  + Optimize data access patterns to minimize cache misses.
  + Investigate potential cache coherency issues in multi-core systems.
* **Floating-Point Arithmetic:**
  + Analyze the use of floating-point arithmetic.
  + Check for precision issues or non-deterministic behavior in floating-point calculations.
  + Consider compiler flags for floating-point consistency.

**8. Compiler and Build Configuration:**

* **Compiler Optimizations:**
  + Review compiler optimization settings.
  + Disable specific optimizations to check if they are causing issues.
  + Ensure consistent compiler versions and configurations across builds.
* **Debug Information:**
  + Enable and analyze debug information in the compiled binary.
  + Use tools like GDB (GNU Debugger) to trace the execution and identify the source of crashes.