An E-Book on

Mastering C for Embedded:

300 Essential C Programming

Challenges to Elevate Your Skill

Written By

Yashwanth Naidu Tikkisetty

What do we have here?

We've got a set of 25 problems in each of the following topics.

1. Strings
2. Arrays
3. Pointers
4. Structures and Unions
5. Linked List
6. Stacks and Queues
7. Memory Management
8. Bit Manipulation
9. Searching
10. Sorting
11. Matrix Based
12. Linux Internals

PROBLEMS ON STRINGS:

1) Print Duplicate characters from a string.

What to do (**WTD**): Traverse through the given string and identify characters that appear more than once.

(e.g.: I/P: "apple" ,O/P: "p")

2) Convert a string to its integer representation without using any built-in functions.

WTD: Transform a given string of numeric characters into its corresponding integer representation without relying on built-in methods.

(e.g., "1234" to 1234)

3) Print the first non-repeated character from a string.

WTD: Examine the string and pinpoint the very first character that doesn't repeat elsewhere in the string.

(e.g.: I/P: "swiss" ,O/P: "w")

4) Find the longest palindrome substring in a given string.

WTD: Identify the longest continuous sequence within the string that reads the same backward as forward.

(e.g.: I/P: "babad" ,O/P: "bab")

5) Check if the string contains only digits.

WTD: Determine if all characters in the provided string are numeric digits.

(e.g.: I/P: "1234a" ,O/P: "False")

6) Duplicate characters found in a string.

WTD: Spot all characters in the string that appear more than once and list them.

(e.g.: I/P: "programming" ,O/P: "r","g","m")

7) Check if a string has balanced parentheses.

WTD: Ensure that for every opening bracket, parenthesis, or brace in the string, there's a corresponding closing counterpart, and they are correctly nested.

(e.g.: I/P: "()[{}]" ,O/P: "True")

8) Count the occurrences of a given character in a string.

WTD: Count how many times a specified character appears in a given string.

(e.g.: I/P: "apple",Char: 'p' ,O/P: "2")

9) Check if two strings are anagrams of each other.

WTD: Determine if two provided strings can be rearranged to form each other, meaning they are anagrams.

(e.g.: I/P: "listen" ,O/P: "silent")

10) Reverse words in a given sentence without using any library method.

WTD: Invert the order of words in a given sentence, maintaining the order of characters within each word.

(e.g.: I/P: "Hello Word" ,O/P: "World Hello")

11) Check if two strings are a rotation of each other.

WTD: Verify if one string can be obtained by rotating another string, indicating how many positions were involved in the rotation.

(e.g.: I/P: "abcde" "cdeab" ,O/P: "Rotation: 2L"(Obtaining String B by rotating String A))

12) Check if a given string is a palindrome.

WTD: Ascertain if the provided string reads the same forwards and backwards.

(e.g.: I/P: "radar" ,O/P: "True")

13) Count the number of vowels and constants in a given string.

WTD: Tally the number of vowel and consonant characters in the given string.

(e.g.: I/P: "apple" ,O/P: Vowels: 2, Consonants: 3)

14) Reverse a string using recursion.

WTD: Use a recursive method to invert the order of characters in a string.

(e.g.: I/P: "hello" ,O/P: "olleh")

15) Find all permutations of a string.

WTD: Generate all possible arrangements of the characters from the given string.

(e.g.: I/P: "ab" ,O/P: "ab","ba")

16) Check for Pangram

WTD: Determine if the given string contains every letter of the alphabet at least once. For instance, "The quick brown fox jumps over a lazy dog" is a pangram.

(e.g.: I/P: "Jinxed wizards pluck ivy from the big quilt"; O/P: True)

17) String Interleaving:

WTD: Determine if a given string is an interleaving of two other strings. For example, "abc" and "123" can be interleaved as "a1b2c3".

(e.g.: I/P: Strings: "xyz", "789", Interleaved: "x7y8z9"; O/P: True)

18) Longest Substring Without Repeating Characters:

WTD: Find the length of the longest substring without repeating characters. For "abcabcbb", the answer would be 3, because the longest substring without repeating letters is "abc".

(e.g.: I/P: "pwwkew"; O/P: 3)

19) Count Substrings with Equal 0s and 1s.

WTD: Given a binary string, count the number of substrings that have an equal number of 0s and 1s. For "0011", the answer is 4: "00", "11", "0011", and "01".

(e.g.: I/P: "0101"; O/P: 4)

20) Find Lexicographically Minimal Rotation.

WTD: Determine the lexicographically smallest rotation of a string. For "bca", the rotations are "bca", "cab", and "abc", and the smallest is "abc".

(e.g.: I/P: "dacb"; O/P: "acbd")

21) Substring Count.

WTD: Find out how many times a particular substring appears in the given string.

(e.g.: I/P: Main String: "banana", Substring: "ana"; O/P:)

22) String Encoding.

WTD: Encode a string by replacing each character with the third character after it in the alphabet. Wrap around to the start of the alphabet after 'z'.

(e.g.: I/P: "abc"; O/P: "def".)

23) Maximum Occurring Character.

WTD: Determine which character in a string appears the most times.

(e.g.: I/P: "success"; O/P: "s".)

24) Palindrome Partitioning.

WTD: Partition a string such that every substring of the partition is a palindrome. Return the minimum cuts needed.

(e.g.: I/P: "aab"; O/P: 1 (The string can be split as ["aa","b"]).)

25) Repeated Substring Pattern.

WTD: Determine if a given string can be constructed by taking a substring of it and appending multiple copies of the substring together.

(e.g.: I/P: "abab"; O/P: True (Because "ab" is repeated))

PROBLEMS ON ARRAYS:

1) Find the missing number in a given integer array of 1 to 500.

WTD: Examine an array expected to contain consecutive integers from 1 to 500. Identify any integer that is missing from this sequence.

(e.g.: I/P: [1,2,4,5]; O/P: 3)

2) Find the duplicate number on a given integer array.

WTD: Inspect the provided array. Determine if there's any integer that appears more frequently than it should, signifying a duplicate.

(e.g.: I/P: [3,1,3,4,2]; O/P: 3)

3) Find the largest and smallest number in an unsorted integer array.

WTD: Navigate through the elements of the unsorted array, continuously updating the largest and smallest values found to identify the extremities in the array.

(e.g.: I/P: [34, 15, 88, 2]; O/P: Max: 88, Min: 2)

4) Find all pairs of an integer array whose sum is equal to a given number.

WTD: Explore combinations of integer pairs in the array. Check if the sum of any of these pairs matches a specified target number.

(e.g.: I/P: [2,4,3,5,6,-2,4,7,8,9], Sum: 7; O/P: [2,5],[4,3])

5) Find duplicate numbers in an array if it contains multiple duplicates.

WTD: Examine the array to identify numbers that appear more than once. Compile a list of these repetitive numbers.

(e.g.: I/P: [4,3,2,7,8,2,3,1]; O/P: [2,3])

6) Sort an array using quicksort algorithm.

WTD: Implement the quicksort sorting technique on the provided array to rearrange its elements in ascending order.

(e.g.: I/P: [64, 34, 25, 12, 22, 11, 90]; O/P: [11, 12, 22, 25, 34, 64, 90])

7) Remove duplicates from an array without using any library.

WTD: Navigate through the array, identifying and removing any repetitive occurrences of numbers, ensuring each number appears only once.

(e.g.: I/P: [1,1,2,2,3,4,4]; O/P: [1,2,3,4])

8) Determine the intersection of two integer arrays.

WTD: Compare every element of the two arrays, listing down the common integers that appear in both.

(e.g.: I/P: [1,2,4,5,6], [2,3,5,7]; O/P: [2,5])

9) Rotate an array to the right by k steps.

WTD: Modify the array by moving its elements to the right, wrapping them around when they reach the end, for a specified number of steps.

(e.g.: I/P: [1,2,3,4,5], k=2; O/P: [4,5,1,2,3])

10) Count occurrences of a number in a sorted array.

WTD: For a given number and a sorted array, iterate through the array to count the number of times that particular number appears.

(e.g.: I/P: [1, 2, 2, 2, 3], 2; O/P: 3)

11) Find the "Kth" max and min element of an array.

WTD: Sort the array and retrieve the kth largest and kth smallest numbers.

(e.g.: I/P: [7, 10, 4, 3, 20, 15], K=3; O/P: 7)

12) Move all zeros to the left of an array while maintaining the order of other numbers.

WTD: Reorder the array by moving all zero values to the leftmost positions while ensuring the relative order of the non-zero numbers remains unchanged.

(e.g.: I/P: [1,2,0,4,3,0,5,0]; O/P: [0,0,0,1,2,4,3,5])

13) Merge two sorted arrays to produce one sorted array.

WTD: Sequentially compare the elements of two sorted arrays, combining them into a single array that remains sorted.

(e.g.: I/P: [1,3,5], [2,4,6]; O/P: [1,2,3,4,5,6])

14) Find the majority element in an array (appears more than n/2 times).

WTD: Traverse the array and maintain a count of each number. Identify if there's any number that appears more than half the length of the array.

(e.g.: I/P: [3,3,4,2,4,4,2,4,4]; O/P: 4)

15) Find the two repeating elements in a given array.

WTD: Investigate the array and find two numbers that each appear more than once.

(e.g.: I/P: [4, 2, 4, 5, 2, 3, 1]; O/P: [4,2])

16) Rearrange positive and negative numbers in an array.

WTD: Sort the array such that all the positive numbers appear before the negative ones, while maintaining their original sequence.
(e.g.: I/P: [-1, 2, -3, 4, 5, 6, -7, 8, 9]; O/P: [4,-3,5,-1,6,-7,2,8,9])

17) Find if there's a subarray with zero sum.

WTD: Explore the array's subarrays (subsets of consecutive elements) to determine if there exists any subarray that sums up to zero.

(e.g.: I/P: [4, 2, -3, 1, 6]; O/P: True)

18) Find the equilibrium index of an array (where sum of elements on the left is equal to sum on the right).

WTD: Examine the array to find an index where the sum of all elements to its left is equal to the sum of all elements to its right.

(e.g.: I/P: [-7, 1, 5, 2, -4, 3, 0]; O/P: 3)

19)Find the longest consecutive subsequence in an array.

WTD: Examine the array to find the longest stretch of numbers that appear in increasing consecutive order.

(e.g.: I/P: [1, 9, 3, 10, 4, 20, 2]; O/P: [1, 2, 3, 4])

20) Rearrange array such that arr[i] becomes arr[arr[i]].

WTD: Transform the array such that the number at each index corresponds to the number found at the index from the original array specified by the current number.

(e.g.: I/P: [0, 1, 2, 3]; O/P: [0, 1, 2, 3])

21) Find the peak element in an array (greater than or equal to its neighbors).

WTD: Scrutinize the array to find an element that is both larger than its predecessor and its successor.

(e.g.: I/P: [1, 3, 20, 4, 1, 0]; O/P: 20)

22) Compute the product of an array except self.

WTD: For every index in the array, calculate the product of all numbers except for the number at that index.

(e.g.: I/P: [1,2,3,4]; O/P: [24,12,8,6])

23) Compute the leaders in an array.

WTD: Traverse the array from right to left, finding numbers that remain the largest compared to all numbers on their right.

(e.g.: I/P: [16,17,4,3,5,2]; O/P: [17,5,2])

24) Find if an array can be divided into pairs whose sum is divisible by k.

WTD: Examine the array to see if it can be segmented into pairs such that the sum of each pair's numbers is divisible by a specific number, k.

(e.g.: I/P: [9, 7, 5, -3], k=6; O/P: True)

25) Find the subarray with the least sum.

WTD: Investigate all possible subarrays of the given array, finding the one with the smallest sum.

(e.g.: I/P: [3,1,-4,2,0]; O/P: -4)

PROBLEMS ON POINTERS:

1. Write a program to find the size of a data type without using the `sizeof` operator, use pointer arithmetic.

WTD: Declare a pointer to the given type and increment it. Subtract the original pointer value from the incremented value to get the size.

(e.g.: I/P: int; O/P: 4(based on platform))

2. Design a function that determines if two pointers point to the same array.

WTD: Traverse from both pointers in both forward and backward directions until null or boundary is hit. If both pointers hit the same boundaries, they belong to the same array.

(e.g.: I/P: int arr[] = {1,2,3,4}, *ptr1 = &arr[1], *ptr2 = &arr[3]; O/P: True)

3. Create a function that uses pointer arithmetic to count the number of elements in an array without utilizing loop constructs.

WTD: Subtract the pointer to the first element from the pointer just past the last element.

(e.g.: I/P: int arr[] = {1,2,3,4,5} ,O/P: 5)

4. Implement a program that swaps two strings using pointers to pointers.

WTD: Use a pointer to pointer to swap the base addresses of the two strings.

(e.g.: I/P: char *str1 = "hello", *str2 = "world" ,O/P: str1 = "world", str2 = "hello")

5. Create a function that segregates even and odd values of an integer array using pointers.

WTD: Use two pointers, one starting from the beginning and the other from the end. Traverse and swap even and odd numbers until the two pointers meet.

(e.g.: I/P: int arr[] = {12,34,9,8,45,90} ,O/P: {12,34,8,90,9,45})

6. Design a program to concatenate two strings without using standard library functions, only pointers.

WTD: Traverse the first string till the end and then copy the second string from that point.

(e.g.: I/P: char *str1 = "Good ", *str2 = "Morning" ,O/P: "Good Morning")

7. Implement a function that splits a string into two halves and returns pointers to the beginning of each half.

WTD: Use pointer arithmetic to find the middle of the string. Return the original and the middle pointers.

(e.g.: I/P: "HelloWorld" ,O/P: "Hello", "World")

8. Write a function that trims leading and trailing whitespace from a string using pointers.

WTD: Use two pointers to find the first and last non-whitespace characters. Move characters to trim the string

(e.g.: I/P: " Hello World " ,O/P: "Hello World")

9. Design a program to find the overlapping part of two arrays using pointers.

WTD: Use two pointers to traverse both arrays. When a common element is found, move both pointers forward.

(e.g.: I/P: int arr1[] = {1,2,3,4,5,6}, arr2[] = {5,6,7,8} ,O/P: {5,6})

10. Create a function that rotates an array to the right by `k` elements using pointers.

WTD: Reverse the whole array, then reverse the first k elements, and finally reverse the rest.

(e.g.: I/P: int arr[] = {1,2,3,4,5}, k=2 ,O/P: {4,5,1,2,3})

11. Implement a function that merges two sorted arrays into a third array using pointers.

WTD: Use three pointers to traverse and compare the two input arrays and insert the smaller element into the third array.

(e.g.: I/P: int arr1[] = {1,3,5}, arr2[] = {2,4,6} ,O/P: {1,2,3,4,5,6})

12. Design a program that checks if a string is a prefix of another string using pointers.

WTD: Traverse both strings using two pointers. If all characters of the shorter string match the beginning of the longer string, return True.

(e.g.: I/P: char *str1 = "Hello", *str2 = "Hel" ,O/P: True)

13. Write a function that converts a string to lowercase using pointers.

WTD: Traverse the string. For each character, if it's uppercase, add 32 to convert it to lowercase.

(e.g.: I/P: "HELLO" ,O/P: "hello")

14. Implement a program that finds the first non-repeated character in a string using pointers.

WTD: Use a fixed-size array to count occurrences. Traverse the string twice: first to count and then to find the non-repeated character.

(e.g.: I/P: "swiss" ,O/P: 'w')

15. Design a function that uses pointers to find the intersection of two arrays.

WTD: If the arrays are sorted, use two pointers to traverse and find common elements. If not, use a hash table for one array and search for elements of the other.

(e.g.: I/P: int arr1[] = {1,2,3,4}, arr2[] = {3,4,5,6} ,O/P: {3,4})

16. Create a program that checks if a string is a palindrome using pointers.

WTD: Use two pointers, one at the start and the other at the end. Traverse inward, comparing characters.

(e.g.: I/P: "radar" ,O/P: True)

17. Write a function to calculate the dot product of two vectors using pointers.

WTD: Traverse both vectors, multiplying corresponding elements and summing the results.

(e.g.: I/P: int vec1[] = {1,2,3}, vec2[] = {4,5,6} ,O/P: 32)

18. Design a program that finds the length of a linked list using double pointers.

WTD: Use a double pointer (or two pointers) technique. Move one pointer twice as fast as the other. When the faster one reaches the end, the slower one will be halfway.

(e.g.: I/P:1->2->3->4->5 ,O/P:5)

19. Implement a function that finds the common elements in three sorted arrays using pointers.

WTD: Use three pointers. Move the pointer pointing to the smallest value until all three pointers point to the same value or the end is reached.

(e.g.: I/P: int arr1[] = {1,5,10,20,40,80}, arr2[] = {6,7,20,80,100}, arr3[] = {3,4,15,20,30,70,80,120} ,O/P:{20,80})

20. Create a program that flattens a 2D array into a 1D array using pointers.

WTD: Use nested loops (or equivalent pointer arithmetic) to traverse the 2D array and copy each element to the 1D array.

(I/P: int arr[][] = {{1,2},{3,4}} ,O/P:{1,2,3,4})

21. Memory Leak Detector. Implement a custom memory allocator and deallocator. Track allocations and deallocations to detect memory leaks using pointers.

WTD: Implement custom allocation and deallocation functions. Maintain a list of allocated blocks. On deallocation, remove from the list. At the end, non-deallocated blocks are leaks.

(e.g.: I/P: Allocate 5 blocks, deallocate 4 blocks ; O/P: 1 block not deallocated)

22. Create a structure representing a point in 2D space. Implement a function that moves the point using pointer arithmetic.

WTD: Access the x and y coordinates of the point using pointers and modify them based on the move values.

(e.g.: I/P: Point (2,3), Move by (1,-1) ; O/P: New Point (3,2))

23. Implement a custom strtok() function using pointers that splits a string based on a delimiter.

WTD: Use pointers to traverse the string. When the delimiter is found, replace it with '\0' and return the start of the token.

(e.g.: I/P: String = "embedded,systems,linux", Delimiter = "," ; O/P: "embedded", "systems", "linux")

24. Design a function similar to memcpy() that copies memory areas using pointers.

WTD: Use pointers to traverse both the source and destination and copy each byte.

(e.g.: I/P: Source = "hello", Destination (empty buffer), Length = 5 ; O/P: Destination = "hello")

25. Implement a function that detects if a pointer has gone out of bounds of an array.

WTD: Compare the pointer with the address of the first and past-the-last elements of the array.

(e.g.: I/P: Array = {1,2,3,4,5}, Pointer pointing after last element ; O/P: Out of bounds)

PROBLEMS ON STRUCTURES AND UNIONS:

1) Define a structure to represent a 3D point in space. Write functions to calculate the distance between two points.

WTD: Design a structure to model a 3D point in space. Develop functions that calculate the Euclidean distance between any two given points using the standard distance formula. Use Distance formula D= Sqrt((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2))

(e.g: I/P P1(1.0,2.0,3.0), P2(4.0,6.0,8.0); O/P is Distance= 7.071)

2) Define a structure for a student with name, roll number, and marks in 5 subjects. Calculate the average marks for a list of students. (Structure with Arrays).

WTD: Construct a structure that encapsulates a student's details, specifically their name, roll number, and scores in five subjects. Implement a function that computes the average marks for an array of student structures.

(e.g: I/P: Name: "Alex", Roll: 105, Marks: [75, 80, 88, 82, 86]; O/P: Avg Marks: 82.2)

3) Create a structure for a book with title, author, and price. Implement a function to discount the book's price by a given percentage.

WTD: Design a function that applies a specified discount percentage to the book's price, updating its value accordingly.

(e.g: I/P: Title: "Pride and Prejudice", Author: "Austen", Price: $30, Discount: 15%; O/P: New Price: $25.5)

4) Design a structure for an employee with an embedded structure for address (street, city, state, zip). (Nested Structure)

WTD: Within this structure, embed another structure specifically meant for the employee's address details, capturing street, city, state, and zip code. Ensure capabilities to extract and display this address in a coherent format.

(e.g: I/P: Name: "Bob", Address: [Street: "456 Maple Rd", City: "Brookfield", State: "WI", Zip: "53005"]; O/P: Address: 456 Maple Rd, Brookfield, WI, 53005)

5) Use a union to represent a 32-bit value that can be accessed as either two 16-bit values or four 8-bit values.

WTD: Implement a union that can hold a 32-bit numerical value. This union should allow for access to the stored number as either two separate 16-bit values or as four distinct 8-bit values.

(e.g: I/P: Value: 0xABCD1234; O/P: 16-bits: 0xABCD, 0x1234; 8-bits: 0xAB, 0xCD, 0x12, 0x34)

6) Define a structure and determine its memory size. Rearrange its members to minimize memory wastage due to alignment. (Structure Memory Alignment)

WTD: Construct a structure and evaluate its memory consumption. Reorganize the structure's components in a manner that minimizes memory wastage due to alignment constraints inherent in system architecture.

(e.g: I/P: Structure: int, char, short; O/P: Memory: 11 bytes; Optimized: 10 bytes)

7) Implement a mini database for a library using an array of structures. Include functionalities like adding a new book and searching for a book by title. (Array of Structures)

WTD: Develop a basic book database for a library using an array of structures. This database should support operations like the addition of new books and title-based book searches.

(e.g: I/P: Add: "To Kill a Mockingbird", Search: "Mockingbird"; O/P: Found: "To Kill a Mockingbird")

8) Create a linked list of students using structures and dynamic memory allocation. (Dynamic Memory with Structures)

WTD: Implement a linked list that represents student records. For this purpose, use structures combined with dynamic memory allocation techniques. Ensure the capability to append new student records to the list.

(e.g: I/P: Add Student: "Mia", Roll: 110; O/P: Student Mia, Roll: 110 Added)

9) Use a union to interpret a 4-byte array as an integer and a floating-point number.

WTD: Design a union that is capable of holding a 4-byte array. This union should facilitate the interpretation of this array in two ways: as an integral value and as a floating-point number.

(e.g: I/P: Bytes: [0x43, 0x48, 0x00, 0x00]; O/P: Float: 134.0)

10) Define a structure with bit fields to represent a set of configurations/settings for a device. (Bit Fields for compact Structure)

WTD: Construct a structure with bit fields to efficiently represent a device's varied configurations or settings in a compact manner.

(e.g: I/P: Config: 1001; O/P: Setting 1 & 4: ON)

11) Implement a function that takes two date structures (day, month, year) and returns the difference in days. (Passing Structures to Functions)

WTD: Develop a function that accepts two date structures, each detailing the day, month, and year. This function should compute and return the difference between these dates in terms of days.

(e.g: I/P: Date1: 15/02/2020, Date2: 20/02/2020; O/P: Diff: 5 days)

12) Create a union that can hold an IP address as a string and as four separate byte values. Implement functions to convert between the two representations.

WTD: Implement a union capable of storing an IP address. This union should support two representations of the IP: as a singular string and as its four byte-wise components. Design functions that facilitate conversions between these representations.

(e.g: I/P: String: "10.0.0.1"; O/P: Bytes: 10, 0, 0, 1)

13) Create a structure representing a menu item with a name (string) and an associated function pointer to execute when the menu item is selected.(Function pointer in structures)

WTD: Define a structure that models a menu item. Each menu item should have a name (a string) and an associated function pointer. This function pointer should point to the action

to be executed when the menu item is selected.

(e.g: I/P: Select: "Option2"; O/P: "Option2 Executed")

14) Define a structure for a network packet, which includes an enum to represent packet type (e.g., DATA, ACK, NACK). (Enum with structures)

WTD: Design a structure to represent a network packet. This structure should incorporate an enum to denote the packet type, which could be values like DATA, ACK, or NACK.

(e.g: I/P: Packet: ACK; O/P: This is an ACK packet)

15) Design a structure to represent a command and its associated parameters. Implement a function to parse a string command into this structure. (Structure for Command Parsing)

WTD: Formulate a structure that captures a command and any associated parameters. Introduce a function capable of parsing a string-based command and populating this structure with the relevant details.

(e.g: I/P: String: "JUMP 20"; O/P: Command: JUMP, Param: 20)

16) Create a base structure for a vehicle with attributes like weight and max_speed. Extend this to derive structures for a car (with attributes like seating capacity) and a truck (with attributes like max_load) (Structure Inheritance)

WTD: Establish a base structure to depict a vehicle, detailing attributes like its weight and maximum speed. Extend this base to derive specialized structures for different vehicle types like cars (with attributes like seating capacity) and trucks (with features like maximum load capacity).

(e.g: I/P: Truck, Weight: 8000kg, Speed: 120km/h, Load: 5000kg; O/P: Can load up to 5000kg)

17) Define a structure for a complex number. Implement functions for addition, subtraction, multiplication, and division of two complex numbers.

WTD: Design a structure to represent a complex number. Develop a suite of functions that allow for mathematical operations on complex numbers, including addition, subtraction, multiplication, and division.

(e.g: I/P: Complex1: 4+5i, Complex2: 2+3i, SUB; O/P: Result: 2+2i)

18) Use a union to convert endianness (big-endian to little-endian and vice versa) of an integer.

WTD: Implement a union that can hold an integer value. Use this union to facilitate endianness conversion, toggling between big-endian and little-endian representations of the integer.

(e.g: I/P: Int: 0xAABBCCDD; O/P: Converted: 0xDDCCBBAA)

19) Define a structure for representing time (hours, minutes, seconds). Implement functions to add time durations and convert this duration to seconds.

WTD: Define a structure that models time, detailing hours, minutes, and seconds. Introduce functions that can sum two time durations and also convert a given duration into its equivalent representation in seconds.

(e.g: I/P: Time1: 1h 20m, Time2: 0h 50m; O/P: Sum: 2h 10m, Seconds: 7800s)

20) Design a union that can represent an error code as both an integer and a descriptive string.

WTD: Design a union capable of representing an error code in two formats: as a numerical integer and as a descriptive string detailing the error's nature.

(e.g: I/P: Code: 500; O/P: Description: "Internal Server Error")

21) Create a union to hold an IP address both as a string (like "192.168.1.1") and as four separate byte values.

WTD: Create a union that can house an IP address. This union should support dual representations: as a singular cohesive string and as its constituent byte values.

(e.g: I/P: IP String: "10.0.2.15"; O/P: Octets: [10, 0, 2, 15])

22) Create a structure to simulate file attributes like name, type (using enum), size, and creation date (use the Date structure from problem 2).

WTD: Implement a structure that simulates file attributes. This structure should capture details like the file's name, its type (represented using an enum), its size in bytes, and its creation date (leveraging a previously defined Date structure).

(e.g: I/P: Name: "document.txt", Type: Document, Size: 1024 bytes, Creation Date: 2023-08-29; O/P: File: "document.txt", Type: Document, Size: 1024 bytes, Created on: 2023-08-29)

23) Create a structure to represent serialized data packets with fields like header, payload, and checksum. Implement functions to serialize and deserialize data.

WTD: Construct a structure that represents serialized data packets, detailing elements like header, payload, and checksum. Develop accompanying functions that enable the serialization of this structured data into a string and its subsequent deserialization back into the structure.

(e.g: I/P: Header: 0x1234, Payload: [0xAA, 0xBB, 0xCC], Checksum: 0xABCD; O/P: Serialized: [0x12, 0x34, 0xAA, 0xBB, 0xCC, 0xAB, 0xCD])

24) Implement a structure that can store an array of data. However, the data type can vary - it could be an array of integers, floats, or characters. Use a union to manage this variability.

WTD: Design a structure with the ability to store an array of data. To accommodate different data types for the array (like integers, floats, or characters), use a union. Ensure mechanisms to correctly identify and retrieve the stored data type.

(e.g: I/P: DataType: Integer, Data: [1, 2, 3, 4, 5]; O/P: Array of Integers: [1, 2, 3, 4, 5])

25) Create a structure that can represent a color in both RGB and CMYK formats. Use a union to switch between the RGB representation and the CMYK representation.

WTD: Formulate a structure capable of representing color information. This structure should support dual representations: in RGB format and in CMYK format. Using a union, ensure seamless switching between these two representations.

(e.g: I/P: Color Type: RGB, RGB: [255, 0, 128]; O/P: RGB Color: [255, 0, 128])

PROBLEMS ON LINKED LIST:

1) Find Middle element of a singly linked list in one pass.

WTD: Use two pointers, one moving twice as fast as the other, to find the middle element in a single pass.

(e.g.: I/P: 1->2->3->4->5; O/P: 3)

2) Find the length of a singly linked list.

WTD: Traverse the list from head to tail, incrementing a counter to find its length.

(e.g.: I/P: 1->2->3->4; O/P: 4)

3) Reverse a linked list.

WTD: Traverse the list while reversing the next pointers of each node.

(e.g.: I/P: 1->2->3; O/P: 3->2->1)

4) Reverse a singly linked list without recursion.

WTD: Use iterative method to reverse the next pointers of each node.

(e.g.: I/P: 1->2->3; O/P: 3->2->1)

5) Remove duplicate nodes in an unsorted linked list.

WTD: Use a hash table to record the occurrence of each node while traversing the list to remove duplicates.

(e.g.: I/P: 1->2->2->3; O/P: 1->2->3)

6) Find nth node from end of a singly linked list.

WTD: Use two pointers, move one n nodes ahead, then move both until the first one reaches the end.

(e.g.: I/P: 1->2->3->4 (n=2); O/P: 3)

7) Move last element to the front of a given linked list.

WTD: Find the last node and its previous node, change their pointers to move the last node to the front.

(e.g.: I/P: 1->2->3->4; O/P: 4->1->2->3)

8) Delete alternate nodes of a linked list.

WTD: Traverse the list and remove every alternate node.

(e.g.: I/P: 1->2->3->4; O/P: 1->3)

9) Pairwise swap elements of a linked list.

WTD: Swap every two adjacent nodes by adjusting their pointers.

(e.g.: I/P: 1->2->3->4; O/P: 2->1->4->3)

10) Check if a given linked list contains a cycle and what would be the starting node?

WTD: Use Floyd's cycle-finding algorithm to detect the cycle and then find its starting node.

(e.g.: I/P: 1->2->3 (3 points back to 1); O/P: True)

11) Intersection point of two linked lists.

WTD: Use two pointers, one for each list, and traverse to find the intersection point.

(e.g.: I/P: 1->2->3 & 4->5->3; O/P: 3)

12) Segregate even and odd nodes in a linked list.

WTD: Use two pointers to rearrange nodes such that all even and odd elements are together.

(e.g.: I/P: 1->2->3->4; O/P: 2->4->1->3)

13) Merge two sorted linked lists.

WTD: Use a temporary dummy node to hold the sorted list, compare each node and attach the smaller one to the dummy.

(e.g.: I/P: 1->3->5 & 2->4->6; O/P: 1->2->3->4->5->6)

14) Add two numbers represented by linked lists.

WTD: Traverse both lists, sum the corresponding nodes, and manage the carry.

(e.g.: I/P: 2->4 & 5->6 (24 + 56); O/P: 8->0)

15) Find sum of two linked list using stack.

WTD: Use two stacks to hold the numbers from each list, then pop and sum them, storing the result in a new list.

(e.g.: I/P: 2->4 & 5->6 (24 + 56); O/P: 8->0)

16) Compare two strings represented as linked lists.

WTD: Traverse both lists, comparing each node's value. If they are equal throughout, the lists are equal.

(e.g.: I/P: 'a'->'b'->'c' & 'a'->'b'->'c'; O/P: Equal)

17) Clone a linked list with next and random pointer.

WTD: Create a deep copy of the linked list including the random pointers using a hash table to map original nodes to their copies.

(e.g.: I/P: 1->2->3 (random pointers set randomly); O/P: Cloned list with same structure and random pointers)

18) Merge sort on a linked list.

WTD: Implement the Merge Sort algorithm on a linked list, splitting the list into halves and merging them back in sorted order.

(e.g.: I/P: 3->1->2; O/P: 1->2->3)

19) Detect and remove loop in a linked list.

WTD: Use Floyd's algorithm to detect the loop and then remove it by setting the next pointer of the last node in the loop to NULL.

(e.g.: I/P: 1->2->3 (3 points back to 1); O/P: 1->2->3)

20) Flatten a multi-level linked list.

WTD: Use a stack or recursion to flatten the list so that all nodes are at the same level.

(e.g.: I/P: 1->2->3 (2 has child 4->5); O/P: 1->2->4->5->3)

21) Partition a linked list around a given value.

WTD: Traverse the linked list, creating two separate lists - one for values less than the partition value and another for values greater than or equal to the partition value. Finally, merge these lists.

(e.g.: I/P: 1->4->3->2->5->2, Partition Value: 3; O/P: 1->2->2->4->3->5)

22) Remove all nodes in a linked list that have a specific value.

WTD: Traverse the linked list and remove any node that has a value matching the specified value. Make sure to properly update the next pointers and free any removed nodes.

(e.g.: I/P: 1->2->6->3->4->5->6, Value to Remove: 6; O/P: 1->2->3->4->5)

23) Convert a binary number represented by a linked list to an integer.

WTD: Traverse the linked list and convert the binary number represented by the linked list nodes to an integer. Use bit manipulation for the conversion.

(e.g.: I/P: 1->0->1; O/P: 5)

24) Find the common ancestor of two nodes in a binary tree represented as a doubly linked list.

WTD: Traverse the binary tree represented as a doubly linked list and find the common ancestor of the given two nodes. Utilize parent pointers in the doubly linked list to backtrack.

(e.g.: I/P: Nodes 6 and 9 in Binary Tree 4->5->6->7->8->9; O/P: 7)

25) Determine if a linked list is a palindrome.

WTD: Use a slow and fast pointer to find the middle of the list. Reverse the second half and compare it with the first half to determine if the linked list is a palindrome.

(e.g.: I/P: 1->2->2->1; O/P: True)

PROBLEMS ON STACKS AND QUEUES:

1) Byte-Packing Stack.

WTD: Design a stack that efficiently stores 8-bit data in a continuous memory space. Ensure that the 32-bit words are packed without any wastage.

(e.g: I/P: Push 0x01, 0x02, 0x03, 0x04 ; O/P: Memory content - 0x04030201)

2) String Message Queue.

WTD: Implement a queue that specializes in storing and retrieving string messages in a FIFO manner.

(e.g: I/P: Enqueue "HELLO", Enqueue "WORLD" ; O/P: Dequeue - "HELLO", Dequeue - "WORLD")

3) Nested Statement Counter.

WTD: Use a stack to simulate the nested structure of programming constructs like loops or if-statements, and return their depth.

(e.g: I/P: { { } } ; O/P: Depth - 2)

4) Expression Validator.

WTD: Develop a stack-based mechanism to validate the correctness of arithmetic expressions by checking for balanced parentheses and proper operator placement.

(e.g: I/P: "(a+b) * (c-d)" ; O/P: Valid Expression)

5) Command Parser.

WTD: Develop a command parser using a stack that can handle nested commands. The parser should be able to distinguish between different types of commands and their nesting levels. Implement a mechanism to return the depth of the nested commands for debugging or other purposes.

(e.g: I/P: "{CMD1 {CMD2}}" ; O/P: Depth - 2)

6) Palindrome Checker using Stack.

WTD: Create a function that uses a stack to determine whether a given string is a palindrome. Push each character of the string onto a stack and then pop them off to compare with the original string.

(e.g: I/P: "RACECAR" ; O/P: Palindrome)

7) Queue-based Sequence Generator.

WTD: Design a queue-based system that can generate the Fibonacci sequence up to n numbers. The queue should be used to store intermediate Fibonacci numbers and help in generating subsequent numbers in the sequence.

(e.g: I/P: n = 5 ; O/P: 0, 1, 1, 2, 3)

8) Function Call Logger.

WTD: Implement a stack that logs function calls during the runtime of a program. This stack should allow for backtracking, enabling the user to trace the sequence of function calls and understand the flow of execution.

(e.g: I/P: Call FuncA, Call FuncB, Return ; O/P: Current function - FuncA)

9) Queue-based Text Filter.

WTD: Develop a queue-based text filter that removes specific words from a given text string. The words to be filtered out should be enqueued and then compared against the text for filtering.

(e.g: I/P: Text - "Hello world", Filter - "world" ; O/P: "Hello")

10) Recursive to Iterative Converter.

WTD: Create a function that uses a stack to convert a recursive algorithm into its iterative version. For example, convert a recursive Fibonacci function into an iterative one that uses a stack for storage.

(e.g: I/P: Fibonacci(5) ; O/P: 5)

11) Stack-based Text Editor.

WTD: Design a text editor that uses a stack to implement basic text editing features like undo and redo. Each operation should push the current state of the text onto the stack, allowing for easy rollback or re-application of changes.

(e.g: I/P: Add "Hello", Undo, Add "Hi" ; O/P: "Hi")

12) Queue-based Logger.

WTD: Build a queue-based logging system that logs and retrieves various system events. The queue should have a predefined size, and when it gets full, the oldest log entry should be removed to make space for a new one.

(e.g: I/P: Log "Event1", Log "Event2", Retrieve ; O/P: "Event1")

13) Multi-stack Array.

WTD: Design a system that allows multiple stacks to be stored within a single array. The space utilization should be optimized so that as one stack grows, it can acquire more space without causing overflow errors for the other stacks.

(e.g: I/P: Push 1 to Stack1, Push 'a' to Stack2 ; O/P: Array - [1, 'a'])

14) Circular Queue.

WTD: Develop a circular queue that overwrites the oldest elements when the queue reaches its capacity. Implement wrap-around functionality to make sure that new elements are inserted at the start of the array when the end is reached.

(e.g: I/P: Enqueue 1, 2, 3, 4 (Size=3) ; O/P: 2, 3, 4)

15) Min-Element Stack.

WTD: Construct a stack that can return the minimum element from the stack in constant time O(1). Use an auxiliary stack or any other data structure to keep track of the minimum element.

(e.g: I/P: Push 4, Push 2, Push 8 ; O/P: Min - 2)

16) Stack Sorting.

WTD: Design a method to sort a stack. You are allowed to use only one additional stack as a temporary storage. No other data structures should be used.

(e.g: I/P: Stack - [4, 3, 1] ; O/P: Stack - [1, 3, 4])

17) Queue from Stacks.

WTD: Implement a queue using two stacks. Use one stack for enqueuing and another for dequeuing. Make sure the oldest element gets dequeued.

(e.g: I/P: Enqueue 1, Enqueue 2 ; O/P: Dequeue - 1)

18) Stack-based Calculator.

WTD: Build a simple calculator to evaluate postfix expressions. Use a stack to keep track of operands and apply operators as they appear.

(e.g: I/P: "5 1 2 + 4 * + 3 -" ; O/P: 14)

19) Priority Queue using Heap.

WTD: Create a priority queue using either a max-heap or a min-heap. Implement methods for insertion and removal of elements based on their priority.

(e.g: I/P: Insert 3, Insert 1, Insert 4 ; O/P: RemoveMax - 4)

20) Radix Sort using Queue.

WTD: Implement Radix Sort using a queue. Use additional queues to sort each digit from the least significant to the most significant.

(e.g: I/P: [170, 45, 75, 90, 802, 24, 2, 66] ; O/P: [2, 24, 45, 66, 75, 90, 170, 802])

21) Queue with Two Priorities.

WTD: Design a queue data structure that handles two levels of priority (high and low). Ensure that elements with high priority are dequeued before those with low priority.

(e.g: I/P: Enqueue 1 (High), Enqueue 2 (Low) ; O/P: Dequeue - 1)

22) Undo and Redo Stack.

WTD: Implement a system that allows for undo and redo operations. Use two stacks to keep track of all states, one for undo and another for redo.

(e.g: I/P: Write "Hello", Undo, Write "Hi" ; O/P: "Hi")

23) Post-order Traversal using Stack.

WTD: Implement post-order traversal of a binary tree using a stack. Make sure to visit the left subtree, then the right subtree, and finally the root node.

(e.g: I/P: Tree - 1->2->3 ; O/P: 2, 3, 1)

24) Balanced Parentheses using Stack.

WTD: Check for balanced parentheses in a given expression. Use a stack to keep track of opening and closing brackets, braces, and parentheses.

(e.g: I/P: "{[()]}" ; O/P: Balanced)

25) Queue-based Cache.

WTD: Implement a caching mechanism using a queue. When the cache is full, evict the least recently used item.

(e.g: I/P: Cache(2), Put 1, Put 2, Get 1, Put 3 ; O/P: Cache - [1, 3])

PROBLEMS ON MEMORY MANAGEMENT:

1) Stack-based Memory Allocator:

Design a memory allocator that allocates memory in a Last-In-First-Out (LIFO) manner from a fixed-size buffer. The allocator should have allocate and free functions.

(e.g.: I/P: Request for 10, 20, and 15 bytes, then Free 15 bytes; O/P: Addresses allocated and next available address after freeing)

How to do?

- Create a fixed-size buffer (array) to act as the memory pool.
- Keep track of the top of the stack.
- When allocating memory, move the stack top accordingly and return the address.
- When freeing memory, validate that it is indeed the most recently allocated block and then move the stack top back.

Detailed Example: For an allocation request of sizes 10, 20, and 15 bytes, the stack top should move each time memory is allocated. When you free 15 bytes, the stack top should move back to the end of the 20-byte block

Hint: Use a structure to represent each allocation, storing the size and the starting address.

2) Memory Leak Detector:

Implement a simple memory leak detector for dynamic memory allocation functions like malloc and free.

(e.g.: I/P: Allocate 5, Allocate 7, Free 5; O/P: Leak Detected at 7)

How to do?

- Keep a list of all currently allocated blocks.
- When malloc is called, add the block to the list.
- When free is called, remove the block from the list.
- At the end of the program, check if the list is empty. If not, report a leak.

Hint: Use a linked list to keep track of allocated blocks.

Detailed Example:

If you allocate 5 and 7 bytes but only free the 5-byte block, the program should report a memory leak at 7 bytes.

3) Dynamic Array:

Implement a dynamic array with the ability to resize itself.

How to do?

- Start with an initial size.
- When adding an element, check if the array is full. If so, resize it.
- When removing an element, consider resizing down if the array is less than half full.

Hint: Use malloc and free (or your custom allocator) to manage the dynamic memory.

4) Fragmentation Detector:

Write a program that simulates memory allocation and deallocation, and can report the amount of fragmented memory.

(e.g.: I/P: Allocate 10, Free 5, Allocate 3; O/P: Fragmentation = 2 bytes)

How to do?

- Start with a fixed-size memory pool.
- Keep track of allocated and free blocks.
- Fragmentation can be calculated as the total size of free blocks - the largest free block.

Hint: A sorted list or tree can be useful for keeping track of free blocks to quickly find the maximum free block size.

Detailed Example:

If you allocate 10 bytes and then free 5 bytes, followed by allocating 3 bytes, you'd have 2 bytes that couldn't be used for a request of size 3 or more, hence 2 bytes are fragmented.

5) Memory-Mapped Circular Buffer:

Simulate a memory-mapped I/O device by implementing a circular buffer. Implement read and write functions to the buffer, taking care not to overflow.

(e.g.: I/P: Write 5, 7, 9; Read 2 times; Write 12; O/P: Read 5, 7; Next write at address wrapping to start)

How to do?

- Implement a circular buffer with a fixed size.
- Implement read and write functions.
- Keep track of head and tail pointers.
- Make sure to handle the buffer wrap-around condition.

Hint: When the head reaches the end of the buffer, it should wrap around to the beginning. The same applies to the tail.

Detailed Example:

If you write 5, 7, 9 to the buffer and then read twice, you should read 5 and 7. The next write should wrap around if the buffer is full.

6) Memory-bound Priority Queue:

Create a priority queue that only uses a pre-defined array. No extra memory allocations allowed. High-priority items come out first when removed.

(e.g.: I/P: Insert 5, 9, 2; Remove; O/P: Removed 9)

How to Do:

- Declare an array of a fixed size, say N, to represent your priority queue.
- Use an integer variable to keep track of the current size of the queue.
- Implement an insert function that adds elements to the array in such a way that the highest priority element is always at the top (or bottom).
- Implement a remove function that removes and returns the highest priority element.

Detailed Example:

You could use a binary heap data structure. Insert elements using heapify-up and remove elements using heapify-down algorithms.

Hint: If using a binary heap, understand how to maintain the heap properties during insertion and removal.

7) Memory Overlay Management:

Simulate a memory overlay mechanism where different data can occupy the same memory region at different times.

(e.g.: I/P: Load Overlay 1, Load Overlay 2; O/P: Memory region updated)

How to Do:

- Declare a fixed-size array to represent your memory block.
- Create a function to "load" an overlay into this memory block.
- Create a function to "unload" or clear the current overlay.
- Use a variable to keep track of which overlay is currently loaded.

Detailed Example:

Your "load" function could simply copy the contents of an overlay into the array, effectively overwriting whatever was there before.

Hint: Keep state information to know which overlay is currently loaded, and manage transitions properly.

8) Custom strdup Function:

Implement a custom strdup function that copies a string into dynamically allocated memory.

(e.g.: I/P: "hello"; O/P: Address of new string "hello")

How to Do:

- Use strlen to find the length of the input string.
- Use malloc to dynamically allocate memory of length strlen(input) + 1.
- Use strcpy or a loop to copy the contents of the input string into this new memory block.
- Return the address of the new string.

Detailed Example:

You'll need to allocate strlen(input) + 1 bytes to also store the null-terminator.

Hint: Remember to allocate an extra byte for the null-terminator.

9) Memory Alignment Checker:

Write a function that checks if a pointer is aligned to a specified number of bytes.

(e.g.: I/P: Address 0x1004, Alignment 4; O/P: Aligned)

How to Do:

- Take the memory address and the alignment size as inputs.
- Use bitwise AND between (address & (alignment - 1)). If the result is 0, the memory is aligned.

Detailed Example:

If the address is 0x1004 and the alignment is 4, then (0x1004 & (4 - 1)) == 0, which means it is aligned.

Hint: The bitwise AND operation is your main tool here.

10) Memory Initialization Library:

Create a library function to initialize a block of memory with a specific value.

(e.g.: I/P: Block size 5, Value 0xAA; O/P: 0xAAAAAAAAAA)

How to Do?

- Accept the block size and the value to initialize with as parameters.
- Use a loop to go through each byte in the block and set it to the specified value.

Detailed Example:

If you have to fill a block of 5 bytes with the value 0xAA, your output memory block would look like 0xAAAAAAAAAA.

Hint: You can use loops or the memset function to accomplish this task.

11) Heap Metadata Inspector:

Implement a function to inspect and print the metadata of the heap, like block sizes and free/used status.

(e.g.: I/P: Allocate 5, Free 2; O/P: Block 1: Used, Block 2: Free)

How to Do?

- Create a simulated heap with an array and metadata structures for each block.
- Upon allocation or freeing, update the block's metadata.
- Create a function to loop through the metadata structures and print out their status.

Detailed Example:

Your metadata might contain fields like isFree and blockSize. When you allocate 5 and then free 2, your function could print out: Block 1: Used, size 5; Block 2: Free, size 2.

Hint: Check out how dynamic memory allocators keep track of block sizes and free/used status for ideas on storing metadata.

12) Object Serialization:

Implement object serialization and deserialization into/from a byte array.

(e.g.: I/P: Object {a: 1, b: 2}; O/P: Byte array)

How to Do?

- Loop through each field in the object and convert it to a byte array.
- Concatenate these byte arrays together for serialization.
- For deserialization, read the byte array in chunks corresponding to each field and convert it back to the object.

Detailed Example:

If your object has integer fields a and b, your byte array after serialization might look like [00 00 00 01 00 00 00 02] for {a: 1, b: 2}.

Hint: Consider using a standard format like JSON or Protocol Buffers as a blueprint for your own simple serialization format.

13) Stack Overflow Detector:

Implement a stack overflow detector for a simulated stack data structure.

(e.g.: I/P: Push 10 items in a stack of size 9; O/P: Stack Overflow)

How to Do?

- Create a stack with a maximum size attribute.
- Implement push and pop methods.
- During each push, check if adding an element would exceed the stack's maximum size. If it would, output "Stack Overflow".

Detailed Example:

If your stack has a maximum size of 9, attempting to push a 10th item should trigger your overflow detection and output "Stack Overflow".

Hint: Use an integer variable to keep track of the current number of elements in the stack.

14) Memory Footprint Analyzer:

Write a program that can analyze the memory footprint of various data structures. Also, don't limit it to known data types. When given a user defined structure or union find its memory size.

(e.g.: I/P: int, float, double; O/P: 4, 4, 8)

How to Do?

- Use the sizeof operator to find the size of various data types like int, float, and double.
- Output these sizes.

Detailed Example:

Simply using sizeof(int) would return 4, sizeof(float) would return 4, and sizeof(double) would return 8 on most systems.

Hint: The sizeof operator will give you the size in bytes, making this fairly straightforward.

15) Write-Once Memory Simulator:

Implement a write-once memory block where each byte can be written only once.

(e.g.: I/P: Write 0xAA to address 5 twice; O/P: Write Failed)

How to Do?

- Create an array to act as your memory block.
- Create a boolean array of the same size to track if a location has been written to.
- When attempting to write, check the boolean array. If the byte has been written to already, output "Write Failed".

Detailed Example:

If you attempt to write 0xAA to address 5 two times, the second attempt should consult the boolean array, find it's already been written to, and output "Write Failed".

Hint: A secondary array can act as flags to keep track of each byte's write status.

16) Memory Access Logger:

Implement a logger that logs each memory access (read or write) to a simulated block of memory.

(e.g.: I/P: Write to address 5, Read from address 5; O/P: Log)

How to Do?

- Create an array to simulate a block of memory.
- Implement a logger function that takes an address and an operation type as inputs.
- Use a data structure like a list or a dictionary to record each read and write operation.
- For each operation, store the address and the type of operation (read or write) in the logger.

Detailed Example:

If you write to address 5 and then read from address 5, the log should contain these two operations.

Hint: You can use an array or list to act as the memory block and another data structure to log the operations.

17) Memory Offset Calculator:

Create a function that calculates the offset of a given field in a structure.

(e.g.: I/P: Field 'x' in struct A; O/P: Offset in bytes)

How to Do?

- Define the structure with its fields.
- Implement a function that takes the name of a field as an argument.
- Use the offsetof macro or manual pointer arithmetic to find the offset of the given field.
- Return the offset in bytes.

Detailed Example:

For a struct with fields 'a', 'b', and 'x', finding the offset of 'x' would return its position in bytes from the beginning of the struct.

Hint: The offsetof macro in C can help find the offset easily.

18) Memory Statistics Reporter:

Create a memory management system that keeps track of allocated and free blocks.

(e.g.: I/P: Allocate 100 bytes, Free 50 bytes; O/P: Total used: 50 bytes, Free blocks: 1, Largest free block: 50 bytes)

How to Do?

- Implement a function to calculate and report statistics.
- For each allocation or deallocation, update your records of used memory and free blocks.
- The function should return the total memory used, the number of free blocks, and the size of the largest free block.

Detailed Example:

If you allocate 100 bytes and then free 50 bytes, the statistics should show 50 bytes in use, 1 free block, and the largest free block being 50 bytes.

Hint: Use a data structure to keep track of allocations and deallocations.

19) Memory Safety Checker:

Write a function to check for buffer overflows and underflows.

(e.g.: I/P: Buffer of 10 bytes, Write 12 bytes; O/P: Buffer overflow detected)

How to Do?

- Implement a function to calculate and report statistics.
- For each allocation or deallocation, update your records of used memory and free blocks.
- The function should return the total memory used, the number of free blocks, and the size of the largest free block.

Detailed Example:

If you try to write 12 bytes into a 10-byte buffer, the function should detect this and output "Buffer overflow detected."

Hint: Always check the size before performing memory operations.

20) Slab Allocator:

Implement a simple slab allocator.

(e.g.: I/P: Allocate object of 32 bytes; O/P: Address from slab allocator)

How to Do?

- Initialize your memory pool with fixed-size blocks, known as slabs.
- Implement a function to allocate memory, which checks for an available slab of the requested size.
- If a slab is available, mark it as used and return its address.
- Implement a function to deallocate memory, which marks a slab as free.
- Optionally, implement a function to merge free slabs and create larger blocks of memory.

Detailed Example:

When you request an object of 32 bytes, the allocator should return an address from its pool of 32-byte slabs.

Hint: You may use linked lists or arrays to manage slabs of memory.

21) Segmented Memory Allocator:

Create multiple fixed-size buffers (arrays) to act as segments.

(e.g.: I/P: Allocate 10 bytes in Segment A, Allocate 20 bytes in Segment B; O/P: Addresses allocated in segments)

How to Do?

- Create an array of structures, where each structure represents a memory segment with its own memory buffer and stack top.
- Implement allocate and free functions that take an additional parameter to specify the segment.
- Inside these functions, use the corresponding segment's stack top to allocate or free memory.
- Update the stack top for the specified segment when memory is allocated or freed.

Detailed Example:

If you allocate 10 bytes in Segment A and 20 bytes in Segment B, each segment's stack top should move accordingly.

Hint: You can use an array of structures, where each structure represents a segment with its own stack top and memory buffer.

22) Garbage Collector Simulator:

Implement a simple mark-and-sweep garbage collector.

(e.g.: I/P: Allocate 5, Allocate 7, Mark 5, Sweep; O/P: 7 collected)

How to Do?

- Create a list or array to keep track of all currently allocated memory blocks.
- Add a boolean field to each block to represent whether it is "marked" for garbage collection.
- Implement a mark function that takes an address and marks the corresponding block.
- Implement a sweep function that frees all unmarked blocks and unmarks the remaining blocks.

Detailed Example:

If you allocate blocks of 5 and 7 bytes and then mark the 5-byte block, running the sweep should collect the 7-byte block.

Hint: Use a linked list or array to keep track of allocated blocks, along with a "marked" flag.

23) Virtual Memory Simulator:

Simulate virtual memory using an array to represent physical memory and another array to represent the page table.

(e.g.: I/P: Access page 3, Access page 7; O/P: Page 3 loaded, Page 7 loaded)

How to Do?

- Use one array to simulate the physical memory and another array to act as the page table.
- Implement an accessPage function that checks if the requested page is in physical memory. If not, it "loads" it.
- Update the page table whenever a page is loaded into physical memory.

Detailed Example:
If you access pages 3 and 7, the simulator should indicate that these pages have been loaded into physical memory.

Hint: The page table will help you map virtual addresses to physical addresses.

24) Memory Pool:

Create multiple memory pools, each with its own fixed-size buffer.

(e.g.: I/P: Allocate 5 from Pool A, Allocate 7 from Pool B; O/P: Addresses allocated from pools)

How to Do?

- Create multiple arrays to serve as different memory pools.
- Each pool should have its own list of free blocks.
- Implement allocate and free functions that take an additional parameter to specify the pool.
- Use the specified pool's free list to allocate and free blocks.

Detailed Example:

If you allocate 5 bytes from Pool A and 7 bytes from Pool B, each pool should return an address from its own buffer.

Hint: You can manage each pool's free list with a linked list or array.

25) Memory Defragmenter:

Implement memory allocation and freeing in a simulated memory pool.

(e.g.: I/P: Allocate 10, Free 5, Defragment; O/P: Memory defragmented)

How to Do?

- Implement a function called defragment in your memory management system.
- This function should go through the list of free blocks and consolidate adjacent free blocks into larger blocks.
- Update your records of free and allocated blocks to reflect this consolidation.

Detailed Example:

If you allocate 10 bytes, free 5 bytes, and then run the defragmenter, it should rearrange memory so that all the free space is contiguous.

Hint: You'll need to update your records of allocated and free blocks during defragmentation.

PROBLEMS ON BIT MANIPULATION:

1. Write a function that toggles the 3rd and 5th bits of an 8-bit number.

WTD: You have an 8-bit number. Your task is to toggle the bits at positions 3 and 5 (counting from the least significant bit). Use bitwise operations to modify these specific bits while leaving the others unchanged.

(e.g.: I/P: 0b10100101; O/P: 0b10000101)

2. Set Bits Without Using Arithmetic Operations. Implement a function that sets the first n bits of a byte to 1.

WTD: You are given an integer n. Use bitwise operations to set the first n bits of an 8-bit number to 1 while setting the remaining bits to 0. You cannot use arithmetic operations like addition or multiplication.

(e.g.: I/P: n=4; O/P: 0b11110000)

3. Detecting Power of Two. Write a program to check if a given number is a power of two using bit manipulation.

WTD: Given an integer, determine whether it is a power of 2 or not. Your solution should only use bitwise operations. Avoid using mathematical functions like logarithms or multiplication.

(e.g.: I/P: 32; O/P: True)

4. Count Set Bits. Design a function that counts the number of set bits (1s) in an integer without looping.

WTD: Given an integer, count the number of bits set to 1 in its binary representation. You need to achieve this without using any loops. Use bitwise operations to find the count efficiently.

(e.g.: I/P: 0b11010110; O/P: 5)

5. Swap Odd and Even Bits. Implement a function to swap odd and even bits in an integer.

WTD: For a given integer, swap its odd and even bits. Bit 1 should swap with Bit 2, Bit 3 with Bit 4, and so on. Use bitwise operations to perform the swapping.

(e.g.: I/P: 0b10101010; O/P: 0b01010101)

6. Single Number in Array. Given an array where every element appears twice except for one, find the element using bitwise operations.

WTD: You are given an array where each element appears exactly twice except for one element, which appears only once. Find that unique element using bitwise operations. Do this in constant space.

(e.g.: I/P: [4,1,2,1,2]; O/P: 4)

7. Implement Bitwise Right Shift. Implement a function to perform a bitwise right shift without using the '>>' operator.

WTD: Given an integer and a shift count, implement the bitwise right shift operation without using the '>>' operator. Your function should return the integer after shifting its bits to the right.

(e.g.: I/P: 0b1101, 2; O/P: 0b0011)

8. Isolate the Rightmost Set Bit. Write a function to isolate the rightmost set bit of an integer.

WTD: Given an integer, isolate the rightmost bit set to 1 in its binary representation. Use bitwise operations to find this bit while setting all other bits to 0.

(e.g.: I/P: 0b10100000; O/P: 0b00100000)

9. Generate All Possible Combinations of n Bits. Design a program that generates all the possible combinations of n bits.

WTD: For a given integer n, generate all possible binary sequences of length n using bitwise operations. The output should be a list of all these combinations.

(e.g.: I/P: n=3; O/P: 000, 001, 010, 011, 100, 101, 110, 111)

10. Reverse Bits in a Byte. Implement a function to reverse the bits in a byte.

WTD: You have an 8-bit byte. Reverse the order of its bits using bitwise operations. For example, if the input byte is 0b11001001, the output should be 0b10010011.

(e.g.: I/P: 0b11001001; O/P: 0b10010011)

11. Check for Alternate Bits. Write a function to check if bits in a given number are in alternate pattern.

WTD: Given an integer, check if the bits in its binary representation alternate between 0 and 1. Use bitwise operations to traverse the bits and perform the check.

(e.g.: I/P: 0b10101010; O/P: True)

12. Extract n Bits. Design a function to extract n bits from a byte starting from a given position p .

WTD: You have a byte, and you need to extract n bits starting from a given position p. Use bitwise operations to isolate these n bits and return them as an integer.

(e.g.: I/P: Byte: 0b10101100, n=3, p=2; O/P: 0b101

13. Count Number of Flips to Convert A to B. Implement a function that counts the number of flips required to convert integer A to integer B .

WTD: You are given two integers A and B. Determine the number of bits you need to flip to convert A into B using bitwise operations.

(e.g.: I/P: A=29 (0b11101), B=15 (0b01111); O/P: 2)

14. Determine if Two Integers Have Opposite Signs. Write a function to determine if two integers have opposite signs using bit manipulation.

WTD: You are given two integers. Use bitwise operations to determine if they have opposite signs. Your function should return a boolean value.

(e.g.: I/P: -4, 5; O/P: True)

15. Mask Certain Bits. Implement a function that masks (sets to zero) all bits except for the first n bits of an integer.

WTD: Given an integer, mask (set to zero) all but the first n bits. Use bitwise operations to perform this masking and return the resulting integer.

(e.g.: I/P: 0b10101111, n=4; O/P: 0b00001111)

16. Rotate Bits. Design a program to rotate bits of a number to the left by k positions.

WTD: You have an integer and a number k. Rotate the bits of the integer to the left by k positions using bitwise operations.

(e.g.: I/P: 0b10110011, k=3; O/P: 0b10011101)

17. Check if Binary Representation of a Number is Palindrome. Write a function to check if the binary representation of a number is a palindrome.

WTD: For a given integer, determine whether its binary representation reads the same forwards and backwards. Use bitwise operations to extract each bit for the check.

(e.g.: I/P: 9 (0b1001); O/P: True)

18. Find Parity of a Number. Implement a function to check if the number of 1s in the binary representation of a given number is even or odd.

WTD: Given an integer, find the parity of its binary representation. Use bitwise operations to count the number of bits set to 1 and determine if it's odd or even.

(e.g.: I/P: 7 (0b0111); O/P: Odd)

19. Implement XOR Without Using XOR Operator. Design a program to perform XOR operation on two numbers without using the XOR operator.

WTD: You have two integers. Implement the XOR operation without using the XOR operator. Use bitwise operations to calculate the result.

(e.g.: I/P: 5, 3; O/P: 6)

20. Find a Unique Number in an Array. Given an array where all numbers appear three times except for one, which appears just once. Find the unique number using bitwise operations.

WTD: Given an array where each number appears exactly three times except for one number, find the unique number that appears just once. Use bitwise operations to find this unique number.

(e.g.: I/P: [6,1,3,3,3,6,6]; O/P: 1)

21) Write a function that clears all the bits from the most significant bit through i (inclusive) in a given number.

WTD: You have an integer. Clear all the bits from the most significant bit to bit i (inclusive). Use bitwise operations to perform this action and return the modified integer.

(e.g. :I/P: 0b11111111, i=3; O/P: 0b00001111)

22) Write a function to calculate a^b Using Bit Manipulation :

WTD: Given two integers a and b, calculate a^b using bitwise operations. Avoid using the power operator or any other arithmetic operations.

(e.g.: I/P: 2, 3; O/P: 8)

23) Write a program to find whether a given number is a multiple of 3 using bit manipulation.

WTD: For a given integer, determine if it is a multiple of 3 using only bitwise operations. Your function should return a boolean value.

(e.g.: I/P: 9; O/P: True)

24) Given an array where every element appears twice except for two numbers. Find those two numbers using bitwise operations.

WTD: You are given an array where every element appears exactly twice except for two numbers. Use bitwise operations to find these two unique numbers.

(e.g.: I/P: [4,1,2,1,2,5]; O/P: 4, 5)

25) Write a function that converts a decimal number to its binary representation using bit manipulation.

WTD: Given a decimal number, convert it into its binary representation using bitwise operations. Your function should return the binary string.

(e.g.: I/P: 5; O/P: 0b101)

PROBLEMS ON SORTING:

1) Optimized Bubble Sort:

WTD: Sort an array using an optimized version of bubble sort.

Bubble Sort: This is a simple sorting algorithm that works by swapping adjacent elements if they are in the wrong order. The optimization stops if no swaps are made during a pass.

(e.g.: I/P: [5, 2, 9, 1, 5, 6]; O/P: [1, 2, 5, 5, 6, 9])

2) Linked List Insertion Sort:

WTD: Sort a singly-linked list using insertion sort.

Insertion Sort: A simple sorting algorithm that works by building a sorted array (or list) one element at a time.

(e.g.: I/P: 4 -> 2 -> 1 -> 5; O/P: 1 -> 2 -> 4 -> 5)

3) Recursive Merge Sort:

WTD: Sort an array using recursive merge sort.

Merge Sort: A divide and conquer algorithm that splits the array into halves, sorts them, and merges them back.

(e.g.: I/P: [38, 27, 43, 3, 9, 82, 10]; O/P: [3, 9, 10, 27, 38, 43, 82])

4) Iterative Quick Sort:

WTD: Sort an array using iterative quick sort.

Quick Sort: Another divide and conquer algorithm that works by selecting a 'pivot' element and partitioning the array around the pivot.

(e.g.: I/P: [3, 1, 4, 1, 5, 9, 2, 6, 5]; O/P: [1, 1, 2, 3, 4, 5, 5, 6, 9])

5) Heap Sort with Binary Max Heap:

WTD: Sort an array using heap sort with a binary max heap.

Heap Sort: Builds a binary max heap and then swaps the root with the last element, decreasing the heap size.

(e.g.: I/P: [3, 19, 1, 14, 8, 7]; O/P: [1, 3, 7, 8, 14, 19])

6) Counting Sort for Negative Numbers:

WTD: Sort an array that includes negative numbers using counting sort.

Counting Sort: Works by counting the occurrences of each unique element and then positioning them in a sorted manner.

(e.g.: I/P: [4, 2, 2, 8, 3, 3, 1, -1, -2]; O/P: [-2, -1, 1, 2, 2, 3, 3, 4, 8])

7) Radix Sort for Floating Points:

WTD: Sort an array of floating-point numbers using radix sort.

Radix Sort: Sorts numbers digit by digit, from the least significant digit to the most.

(e.g.: I/P: [0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51]; O/P: [0.32, 0.33, 0.37, 0.42, 0.47, 0.51, 0.52])

8) Dynamic Gap Shell Sort:

WTD: Sort an array using shell sort with a dynamic gap sequence.

Shell Sort: A generalization of insertion sort that works by comparing elements that are distant rather than adjacent.

(e.g.: I/P: [9, 8, 3, 7, 5, 6, 4, 1]; O/P: [1, 3, 4, 5, 6, 7, 8, 9])

9) Comb Sort:

WTD: Sort an array using comb sort.

Comb Sort: A variant of bubble sort that eliminates turtles, or small values near the end of the list.

(e.g.: I/P: [8, 4, 1, 14, 8, 2, 9, 5]; O/P: [1, 2, 4, 5, 8, 8, 9, 14])

10) In-Place Merge Sort:

WTD: Sort an array using in-place merge sort.

In-Place Merge Sort: A variant of merge sort that sorts the array using a constant amount of extra space.

(e.g.: I/P: [12, 11, 13, 5, 6]; O/P: [5, 6, 11, 12, 13])

11) Tim Sort:

WTD: Sort an array using Tim Sort.

Tim Sort: A hybrid sorting algorithm derived from merge sort and insertion sort.

(e.g.: I/P: [5, 21, 9, 1, 45, 15, 67]; O/P: [1, 5, 9, 15, 21, 45, 67])

12) Bucket Sort:

WTD: Sort floating-point numbers using Bucket Sort.

Bucket Sort: Distributes elements into buckets, sorts individual buckets, and then concatenates them.

(e.g.: I/P: [0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434]; O/P: [0.1234, 0.3434, 0.565, 0.656, 0.665, 0.897])

13) Pigeonhole Sort:

WTD: Sort an array of positive integers using Pigeonhole Sort.

Pigeonhole Sort: Suitable for sorting elements where the range of key values is small.

(e.g.: I/P: [8, 3, 2, 7, 4, 6, 8]; O/P: [2, 3, 4, 6, 7, 8, 8])

14) Cocktail Sort:

WTD: Sort an array using Cocktail Sort.

Cocktail Sort: A variation of Bubble Sort that sorts in both directions on each pass.

(e.g.: I/P: [5, 1, 4, 2, 8, 0, 2]; O/P: [0, 1, 2, 2, 4, 5, 8])

15) Cycle Sort:

WTD: Sort an array in-place using Cycle Sort.

Cycle Sort: An in-place, unstable sorting algorithm that is optimal in terms of the total number of writes to the original array.

(e.g.: I/P: [20, 40, 50, 10, 30]; O/P: [10, 20, 30, 40, 50])

16) Gnome Sort:

WTD: Sort an array using Gnome Sort.

Gnome Sort: Also known as "stupid sort," it iterates through the list to be sorted and swaps adjacent elements if they are in the wrong order.

(e.g.: I/P: [34, 2, 10, -9]; O/P: [-9, 2, 10, 34])

17) Odd-Even Sort:

WTD: Sort an array using Odd-Even Sort.

Odd-Even Sort: A variation of bubble sort that first sorts the odd-indexed numbers and then the even-indexed numbers.

(e.g.: I/P: [3, 7, 4, 9, 5, 2, 6, 1]; O/P: [1, 2, 3, 4, 5, 6, 7, 9])

18) Sleep Sort:

WTD: Sort an array using Sleep Sort.

Sleep Sort: This unusual sort involves creating a separate thread for each element in the input array, where each thread sleeps for an amount of time corresponding to the value of the input element.

(e.g.: I/P: [4, 3, 2, 1]; O/P: [1, 2, 3, 4])

19) Pancake Sort:

WTD: Sort an array using Pancake Sort.

Pancake Sort: This involves reversing portions of the array to sort the elements.

(e.g.: I/P: [23, 15, 17, 30]; O/P: [15, 17, 23, 30])

20) Bitonic Sort:

WTD: Sort an array using Bitonic Sort.

Bitonic Sort: First produces a bitonic sequence and then iteratively sorts that sequence.

(e.g.: I/P: [3, 10, 2, 1, 20]; O/P: [1, 2, 3, 10, 20])

21) Stooge Sort:

WTD: Sort an array using Stooge Sort.

Stooge Sort: A recursive sorting algorithm that sorts first 2/3rd of the array, then last 2/3rd, and again first 2/3rd.

(e.g.: I/P: [2, 4, 5, 3, 1]; O/P: [1, 2, 3, 4, 5])

22) Bead Sort:

WTD: Sort an array of non-negative integers using Bead Sort.

Bead Sort: Also known as "Gravity Sort," it uses a grid to sort integers.

(e.g.: I/P: [5, 1, 4, 2, 8]; O/P: [1, 2, 4, 5, 8])

23) Bogo Sort:

WTD: Sort an array using Bogo Sort.

Bogo Sort: Randomly permutes its input until a sorted sequence is produced.

(e.g.: I/P: [3, 0, 2, 5, -1, 4, 1]; O/P: [-1, 0, 1, 2, 3, 4, 5])

24) Permutation Sort:

WTD: Sort an array using Permutation Sort.

Permutation Sort: Generates all possible permutations of the list and checks each one to see if it's sorted.

(e.g.: I/P: [12, 11, 13, 5, 6]; O/P: [5, 6, 11, 12, 13])

25) Patience Sort:

WTD: Sort an array using Patience Sort.

Patience Sort: Solitaire card game-based sorting. It sorts the array by placing elements into "piles" in ascending order.

(e.g.: I/P: [2, 6, 3, 4, 1]; O/P: [1, 2, 3, 4, 6])

PROBLEMS ON SEARCHING:

1) Search for an element in an unsorted array.

WTD: Create a function that takes an unsorted array and a target element as input. Your task is to find the index of the target element using linear search. You can accomplish this by looping through the array and comparing each element with the target.

Algorithm: Linear search sequentially checks each element until the desired element is found or the array ends.

(e.g.: I/P: [5, 2, 9, 1, 5, 6], 9 ; O/P: 2)

2) Search for an element in a sorted array.

WTD: Implement a function that performs binary search to find a target element in a sorted array. You'll need to start by comparing the target with the middle element and adjust your search range accordingly. Divide and conquer by halving the array.

Algorithm: Binary search divides the array into halves and compares the middle element to the target.

(e.g.: I/P: [1, 2, 3, 4, 5], 4 ; O/P: 3)

3) Implement binary search recursively.

WTD: Create a function that uses recursion to implement binary search on a sorted array. Your function should make a recursive call after halving the array based on the comparison with the target element.

Algorithm: Recursive binary search reduces the problem size by half in each recursive call.

(e.g.: I/P: [1, 2, 3, 4, 5], 3 ; O/P: 2)

4) Implement interpolation search on a sorted array.

WTD: Implement a function that employs interpolation search on a sorted array to find a target element. Use a formula to guess the probable position of the target element before conducting the search.

Algorithm: Interpolation search tries to find the position of the target value by using a probing formula.

(e.g.: I/P: [1, 2, 3, 4, 5], 4 ; O/P: 3)

5) Use exponential search to find an element in a sorted array.

WTD: Develop a function that uses exponential search to find a target element in a sorted array. First exponentially grow the range where the target is likely to reside, and then apply binary search within that range.

Algorithm: Exponential search involves two steps: finding a range where the element is likely to be and applying binary search.

(e.g.: I/P: [1, 2, 3, 4, 5], 3 ; O/P: 2)

6) Implement jump search.

WTD: Write a function to implement jump search on a sorted array. Use a fixed step size to skip ahead and then perform a linear search within the step.

Algorithm: Jump search skips ahead by fixed steps and performs a linear search backward.

(e.g.: I/P: [1, 3, 5, 7, 9], 7 ; O/P: 3)

7) Implement Fibonacci search on a sorted array.

WTD: Create a function that uses Fibonacci search to find a target element in a sorted array. Use Fibonacci numbers to divide the array into unequal parts.

Algorithm: Fibonacci search divides the array into unequal parts based on Fibonacci numbers.

(e.g.: I/P: [1, 2, 3, 4, 5], 2 ; O/P: 1)

8) Implement sentinel search on an unsorted array.

WTD: Write a function that employs sentinel search on an unsorted array. Add a sentinel element to the array to avoid checking boundaries.

Algorithm: Sentinel search uses a sentinel value to avoid the boundary check.

(e.g.: I/P: [5, 2, 9, 1, 5, 6], 1 ; O/P: 3)

9) Implement ternary search on a sorted array.

WTD: Implement a function for performing ternary search on a sorted array. Divide the array into three equal parts and use two midpoints for comparison.

Algorithm: Ternary search divides the array into three parts and compares the key with the two midpoints.

(e.g.: I/P: [1, 2, 3, 4, 5], 1 ; O/P: 0)

10) Search an element in a sorted and rotated array.

WTD: Develop a function that searches for an element in a sorted but rotated array. Use a modified binary search that accounts for the rotation.

Algorithm: Modified binary search.

(e.g.: I/P: [3, 4, 5, 1, 2], 1 ; O/P: 3)

11) Find a peak element in an array.

WTD: Write a function to find a peak element in a given array. Use binary search to find an element that is greater than its neighbors.

Algorithm: Use binary search to find a peak element.

(e.g.: I/P: [1, 3, 5, 4, 2] ; O/P: 5)

12) Find the missing number in an array of 1 to N.

WTD: Create a function that finds the missing number in an array that contains numbers from 1 to N. You can use binary search or sum formula to find the missing number.

Algorithm: Use binary search or mathematical formula.

(e.g.: I/P: [1, 2, 4, 6, 5, 7, 8] ; O/P: 3)

13) Find the first and last occurrence of an element in a sorted array.

WTD: Implement a function that finds both the first and last occurrences of a given element in a sorted array. Use a modified binary search algorithm to find the initial and final positions.

Algorithm: Modified binary search.

(e.g.: I/P: [2, 2, 2, 2, 2], 2 ; O/P: First = 0, Last = 4)

14) Find a fixed point (value equal to index) in a given array.

WTD: Write a function to find a fixed point (an element equal to its index) in a sorted array. Use binary search to efficiently find a fixed point.

Algorithm: Use binary search for a sorted array.

(e.g.: I/P: [-10, -5, 1, 3, 7, 9, 12, 17] ; O/P: 3)

15) Count occurrences of a number in a sorted array.

WTD: Develop a function that counts the occurrences of a given element in a sorted array. Use a modified binary search to find the first and last occurrences, then calculate the count.

Algorithm: Modified binary search to find the first and last occurrences.

(e.g.: I/P: [2, 2, 2, 2, 2], 2 ; O/P: 5)

16) Find the majority element in an array, if it exists.

WTD: Create a function to find the majority element in an array if it exists. Utilize the Boyer-Moore Voting Algorithm or a binary search on a sorted array.

Algorithm: Use the Boyer-Moore Voting Algorithm or binary search on a sorted array.

(e.g.: I/P: [3, 3, 4, 2, 4, 4, 2, 4, 4]; O/P: 4)

17) Search for an element in a bitonic array (an array that is first increasing then decreasing).

WTD: Implement a function to search for an element in a bitonic array (an array that first increases and then decreases). First find the peak element, then conduct a binary search on the increasing and decreasing parts.

Algorithm: First find the peak element, then perform binary search on the increasing and decreasing parts.

(e.g.: I/P: [1, 3, 8, 12, 4, 2]; Search Element: 8; O/P: 2)

18) Search for an element in a matrix where rows and columns are sorted.

WTD: Develop a function to search for an element in a matrix where the rows and columns are sorted. Start from the top-right or bottom-left corner and incrementally move toward the target.

Algorithm: Start from the top-right or bottom-left corner and move towards the target.

(e.g.: I/P: Matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]], Search Element = 5; O/P: (1, 1))

19) Count the number of times a sorted array is rotated.

WTD: Write a function that counts the number of times a sorted array is rotated. Use binary search to identify the pivot element, which indicates the rotation count.

Algorithm: Use binary search to find the pivot element.

(e.g.: I/P: [15, 18, 2, 3, 6, 12]; O/P: 2)

20) Find the Kth maximum and minimum element in an array.

WTD: Implement a function to find the Kth maximum and minimum element in an array. Employ QuickSelect, sorting, or a min-max heap method.

Algorithm: Use QuickSelect, sorting, or min-max heap methods.

(e.g.: I/P: [7, 10, 4, 3, 20, 15], K = 3; O/P: Max = 15, Min = 7)

21) In a sorted array, find two numbers that add up to a specific target sum "N".

WTD: Create a function that, in a sorted array, finds two numbers that add up to a specific target sum "N". Use a two-pointer technique or binary search for an efficient solution.

Algorithm: Use two-pointer technique or binary search.

(e.g.: I/P: [1, 2, 3, 4, 5], N = 9; O/P: (4, 5))

22) In a sorted array of distinct elements, find the smallest missing element.

WTD: Write a function to find the smallest missing element in a sorted array of distinct elements. Use binary search to locate the smallest missing number efficiently.

Algorithm: Use binary search to find the smallest missing number.

(e.g.: I/P: [0, 1, 2, 6, 9]; O/P: 3)

23) Search for an element in a nearly sorted array (elements may be swapped).

WTD: Implement a function to search for an element in a nearly sorted array (where elements may be slightly out of place). Use a modified binary search algorithm.

Algorithm: Modified binary search.

(e.g.: I/P: [2, 1, 3, 5, 4], 2; O/P: 0)

24) Given an array of n unique numbers except for one number which is repeating and one number which is missing, find them.

WTD: Develop a function that identifies one number which is repeating and one which is missing in an array of n unique numbers. Leverage XOR bitwise operation or sorting techniques.

Algorithm: Use XOR bitwise operation or sorting.

(e.g.: I/P: [3, 1, 3]; O/P: Repeating = 3, Missing = 2)

25) Given a sorted array of unknown length, find if an element exists.

WTD: Write a function to find if an element exists in a sorted array of unknown length. Utilize exponential search to define a suitable range for binary search.

Algorithm: Use exponential search to find the high boundary for binary search.

(e.g.: I/P: [1, 2, 3, 4, 5, ...], Element = 5; O/P: 4)

MATRIX BASED PROBLEMS:

1) Implement matrix multiplication for two matrices without utilizing dynamic memory allocation.

WTD: Write a function to perform matrix multiplication. The input will be two matrices A and B. Check if the number of columns in A equals the number of rows in B. If yes, proceed with the multiplication; otherwise, return an error. You'll need nested loops for each matrix.

(e.g.: I/P: Matrix A = [[1, 2], [3, 4]], Matrix B = [[2, 0], [1, 3]] ; O/P: Result = [[4, 6], [10, 12]])

$$IP = \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}2 & 0\\1 & 3\end{bmatrix} \quad OP = \begin{bmatrix}4 & 6\\10 & 12\end{bmatrix}$$

2) Efficiently compute the transpose of a matrix using minimal auxiliary space.

WTD: Implement a function to compute the transpose of a matrix in-place, meaning you should minimize the use of any additional memory. You can swap elements across the diagonal.

(e.g.: I/P: Matrix = [[1, 2], [3, 4]] ; O/P: Transpose = [[1, 3], [2, 4]])

$$IP = \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \quad OP = \begin{bmatrix}1 & 3\\2 & 4\end{bmatrix}$$

3) Represent a sparse matrix efficiently using a compressed format and perform matrix-vector multiplication.

WTD: Use a data structure like a list of lists to store only the non-zero elements along with their row and column indexes. Write a function to multiply this sparse matrix with a given vector. Skip zero multiplications for efficiency.

(e.g.: I/P: Matrix = [[5, 0, 0], [0, 8, 0], [0, 0, 3]], Vector = [1, 2, 3] ; O/P: Result = [5, 16, 9])

4) Rotate a matrix representing grayscale image data by 90 degrees.

WTD: Implement a function that rotates a given 2D array (representing grayscale image data) by 90 degrees. You might want to consider in-place rotation for extra challenge.

(e.g.: I/P: Matrix (2x2) representing an image ; O/P: Rotated Matrix)

5) Compute the power of a square matrix using the "square and multiply" method.

WTD: Implement a function that computes the power of a square matrix using the "square and multiply" method. Instead of naive multiplication, use the method to reduce the number of multiplications.

(e.g.: I/P: Matrix A = [[2, 1], [1, 1]], Power = 3 ; O/P: Matrix raised to power 3)

6) Compute the determinant of a 3x3 matrix without using recursion.

WTD: Write a function to calculate the determinant of a 3x3 matrix without using recursion. You can use the rule of Sarrus for 3x3 matrices.

(e.g.: I/P: Matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] ; O/P: Determinant value)

7) Check if a given matrix is symmetric.

WTD: Implement a function that checks if a matrix is symmetric. A matrix is symmetric if it is equal to its transpose.

(e.g.: I/P: Matrix = [[1, 2, 3], [2, 3, 4], [3, 4, 5]] ; O/P: Symmetric)

8) Extract the diagonal elements of a matrix and store them in an array.

WTD: Write a function that takes a matrix as input and returns an array containing its diagonal elements. The diagonal elements are those where the row index equals the column index.

(e.g.: I/P: Matrix = [[1, 0], [0, 2]] ; O/P: Diagonal = [1, 2])

9) Calculate the sum of each row and column for a matrix.

WTD: Implement a function to calculate and return the sum of each row and each column in a given matrix. You'll need two separate arrays for storing these sums.

(e.g.: I/P: Matrix = [[1, 2], [3, 4]] ; O/P: Row Sums = [3, 7], Column Sums = [4, 6])

10) Normalize a matrix such that all values lie between 0 and 1.

WTD: Write a function to normalize a given matrix such that all its elements lie between 0 and 1. You'll need to find the min and max elements first.

(e.g.: I/P: Matrix = [[10, 20], [30, 40]] ; O/P: Normalized matrix with values between 0 and 1)

11) Compute the rank of a matrix.

WTD: Implement a function to find the rank of a given matrix. You may use row reduction to echelon form and then find the number of non-zero rows.

(e.g.: I/P: Matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] ; O/P: Rank)

12) Invert a 2x2 matrix without using library functions.

WTD: Write a function that computes the inverse of a 2x2 matrix, without using any library functions. Use the formula for the inverse of a 2x2 matrix.

(e.g.: I/P: Matrix = [[1, 2], [3, 4]] ; O/P: Inverse matrix)

13) Implement Strassen's method for matrix multiplication.

WTD: Implement Strassen's algorithm for matrix multiplication. This method breaks down the original matrices into smaller matrices and uses them to perform fewer multiplications.

(e.g.: I/P: Two matrices ; O/P: Multiplied matrix)

14) Calculate the sum of boundary elements of a matrix.

WTD: Write a function to calculate and return the sum of the boundary elements of a given matrix. The boundary elements are those that are either in the first or last row, or in the first or last column.

(e.g.: I/P: Matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] ; O/P: Sum = 28)

15) Check if a given matrix is a Toeplitz matrix (diagonal elements are same).

WTD: Implement a function to check if a given matrix is a Toeplitz matrix. A Toeplitz matrix has equal elements along its diagonals from top-left to bottom-right.

(e.g.: I/P: Matrix = [[6, 7, 8], [7, 6, 7], [8, 7, 6]] ; O/P: Toeplitz)

16) Extract the anti-diagonal elements of a matrix.

WTD: Write a function to extract and return the anti-diagonal elements of a matrix. The anti-diagonal elements are those for which row index + column index equals the size of the matrix minus one.

(e.g.: I/P: Matrix = [[1, 2], [3, 4]] ; O/P: Anti-diagonal = [2, 3])

17) Detect if a smaller matrix pattern exists within a larger matrix.

WTD: Implement a function that takes in two matrices, a large one and a smaller one, and returns the position of the smaller matrix if it is found within the larger one. You may need to use nested loops to check each sub-matrix within the larger one.

(e.g.: I/P: Large Matrix, Small Matrix ; O/P: Position of small matrix in large matrix)

18) Scale a matrix by a given factor.

WTD: Write a function to scale all the elements of a matrix by a given factor. You will have to multiply each element of the matrix by the scaling factor.

(e.g.: I/P: Matrix = [[1, 2], [3, 4]], Factor = 2 ; O/P: Scaled Matrix = [[2, 4], [6, 8]])

$IP = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ $OP = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$

19) Rotate a matrix 90 degrees in place.

WTD: Implement a function to rotate a given matrix by 90 degrees in-place. You can perform the rotation by first transposing the matrix and then reversing each row.

(e.g.: I/P: Matrix = [[1, 2], [3, 4]] ; O/P: Rotated Matrix = [[3, 1], [4, 2]])

20) Traverse a matrix in a zigzag pattern.

WTD: Write a function to return the elements of a matrix traversed in a zigzag pattern. You will need to traverse the matrix row-wise but change direction after each row.

(e.g.: I/P: Matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] ; O/P: Traversal = [1, 2, 4, 7, 5, 3, 6, 8, 9])

$$IP = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad OP = [1 \quad 2 \quad 4 \quad 7 \quad 5 \quad 3 \quad 6 \quad 8 \quad 9]$$

21) Perform Matrix Addition and Subtraction

WTD: Implement functions for both matrix addition and subtraction. Remember that the matrices have to be of the same dimensions for these operations.

(e.g.: I/P: Matrix A = [[1, 2], [3, 4]], Matrix B = [[2, 0], [1, 3]]; O/P: Addition = [[3, 2], [4, 7]], Subtraction = [[-1, 2], [2, 1]])

$$IP: Matrix\ A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\ Matrix\ B = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}\ \ OP:\ Add = \begin{bmatrix} 3 & 2 \\ 4 & 7 \end{bmatrix}, Sub = \begin{bmatrix} -1 & 2 \\ 2 & 1 \end{bmatrix}$$

22) Implement Hadamard Product

WTD: Create a function that calculates the Hadamard product (element-wise multiplication) of two matrices. The matrices need to be of the same dimensions.

(e.g.: I/P: Matrix A = [[1, 2], [3, 4]], Matrix B = [[2, 0], [1, 3]]; O/P: Hadamard Product = [[2, 0], [3, 12]])

23) Convert Matrix to Row-Echelon Form

WTD: Write a function that transforms a given matrix into its Row-Echelon form. Use Gaussian elimination techniques.

(e.g.: I/P: Matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]; O/P: Row-Echelon Form)

24) Find Eigenvalues and Eigenvectors of a 2x2 Matrix

WTD: Implement a function that calculates the eigenvalues and eigenvectors of a 2x2 matrix. Eigenvalues can be found as the roots of the characteristic equation, and eigenvectors can be found by plugging the eigenvalues back into the matrix equation.

(e.g.: I/P: Matrix = [[2, 1], [1, 2]]; O/P: Eigenvalues and Eigenvectors)

25) Calculate Matrix Trace and Norm

WTD: Write functions to calculate the trace (sum of diagonal elements) and the Frobenius norm (square root of the sum of all squares) of a matrix. Iterate through the matrix to sum up the required elements.

(e.g.: I/P: Matrix = [[1, 2], [3, 4]]; O/P: Trace = 5, Norm = sqrt(30))

PROBLEMS ON LINUX INTERNALS:

1) Implement a Simple Shell

WTD: Create a minimal shell in C that can execute basic Linux commands like ls, cd, pwd, etc. Use system calls like fork() and exec() to spawn and execute commands.

(e.g.: I/P: "ls -l"; O/P: Directory listing)

2) Custom malloc() and free()

WTD: Implement your own malloc() and free() functions. Use system calls like sbrk() to manage memory allocation.

3) Implement tail Command

WTD: Create a program to mimic the Linux tail command. Use file I/O operations to read the file in reverse and display lines.

(e.g.: I/P: File path; O/P: Last 10 lines of the file)

4) Implement Priority Inversion Handling

WTD: Simulate Priority Inversion and implement a solution. Use threads with different priorities and mutexes to simulate and resolve priority inversion.

(e.g.: I/P: None; O/P: Fixed Priority Inversion)

5) Custom init Process

WTD: Write a custom init process to replace the default init for embedded Linux. Research how init works and write a C program that performs the necessary initialization steps.

(e.g.: I/P: None; O/P: System booted with custom init)

6) Process Forking and Waiting

WTD: Create a program that forks child processes and waits for them to complete. Use fork() to create child processes and wait() to wait for their completion.

(e.g.: I/P: Number of child processes; O/P: Exit status of children)

7) Semaphore-based Producer-Consumer

WTD: Implement a producer-consumer problem using semaphores. Use POSIX semaphores and threads to implement the producer and consumer.

(e.g.: I/P: Number of items to produce and consume; O/P: Order of production and consumption.)

8) User-Space File Reader

WTD: Read a file from a user-space program using system calls. Use open() and read() system calls to read a file and display its contents.

(e.g.: I/P: File name; O/P: File contents)

9) Thread Creation and Synchronization

WTD: Create multiple threads and synchronize them using mutexes. Use pthread_create() for thread creation and pthread_mutex_lock() for synchronization.

(e.g.: I/P: Number of threads; O/P: Thread execution order)

10) Priority Inversion Problem

WTD: Simulate the priority inversion problem and solve it using priority inheritance. Implement priority inheritance while using mutexes.

(e.g.: I/P: Thread priorities; O/P: Execution order)

11) Filesystem Traversal using opendir and readdir

WTD: Traverse a directory structure and print all filenames. Use opendir() and readdir() to read directories and their contents.

(e.g.: I/P: Root directory; O/P: List of all files and directories)

12) User-Space to Kernel-Space Communication

WTD: Implement a simple user-space to kernel-space communication using procfs. Create a proc file and use it to send and receive messages from a kernel module.

(e.g.: I/P: String message; O/P: Echoed message from kernel-space)

13) Zero-Copy Data Transfer

WTD: Implement a zero-copy data transfer mechanism between two processes using shared memory. Use shared memory mechanisms like mmap() for zero-copy transfer.

(e.g.: I/P: Data to transfer; O/P: Data received)

14) CPU Information Fetcher

WTD: Write a program to fetch and display CPU information from /proc/cpuinfo. Use file I/O to read the /proc/cpuinfo file and parse it.

15) File Copy Program

WTD: Write a program that copies one file to another using low-level file operations. Use open(), read(), and write() system calls to perform the file copy.

(e.g.: I/P: Source and destination file paths; O/P: Copy status)

16) Mutex-based File Access

WTD: Write a multi-threaded program where each thread attempts to write to a file. Use mutexes to ensure that only one thread writes to the file at any given time. Use pthread_mutex_lock() to lock the file before each thread writes to it.

(e.g.: I/P: Thread count; O/P: File with written data)

17) Command-line Argument Parser

WTD: Write a C program that takes various command-line options and arguments and prints them. Use the getopt() library function to parse command-line options and arguments.

(e.g.: I/P: Command-line options and arguments; O/P: Parsed data)

18) Implement watch Command

WTD: Write a program to mimic the Linux watch command. Use system() or fork() and exec() to run the command at regular intervals.

(e.g.: I/P: Command to run, time interval; O/P: Command output at intervals)

19) Implement whois Command

WTD: Write a program to mimic the whois command functionality. Use socket programming to communicate with the whois server.

(e.g.: I/P: Domain name; O/P: Whois record)

20) CPU Scheduling Algorithm

WTD: Implement a basic CPU scheduling algorithm like Round Robin. Use a queue to manage processes and simulate the CPU scheduler.

(e.g.: I/P: Processes and burst times; O/P: Turnaround and waiting times)

21)Implement chmod Command

WTD: Create a program to mimic the Linux chmod command for changing file permissions. Use the chmod() system call to change the file permissions.

(e.g.: I/P: File path and permission code; O/P: Permission change status)

22) Terminal Multiplexer

WTD: Create a basic terminal multiplexer that allows switching between multiple shell instances. Use fork() and exec() to spawn new shells and terminal I/O to switch between them.

(e.g.: I/P: Key commands; O/P: Switched terminal instance)

23) Custom ps Command

WTD: Implement a simplified version of the ps command to list the currently running processes. Read information from /proc to list the currently running processes.

(e.g.: I/P: None; O/P: List of running processes)

24) Custom Signal Handling

WTD: Write a program that custom handles signals like SIGINT and SIGTERM. Use signal() or sigaction() to catch and handle signals.

(e.g.: I/P: Signal type; O/P: Custom message or action upon receiving signal)

25) Implement ping Command

WTD: Write a program to mimic the functionality of the ping command. Use raw sockets and ICMP protocol to implement the ping functionality.

(e.g.: I/P: IP Address or hostname; O/P: Ping statistics)

Written By : **Yashwanth Naidu Tikkisetty**

Happy learning.

Learn together, Grow together.

Follow me to receive updates regarding Embedded Systems.

Connect with me on LinkedIn: https://www.linkedin.com/in/t-yashwanth-naidu/

T Yashwanth Naidu

https://github.com/T-Yashwanth-Naidu