1. Recursion

Recursion is a process where a function calls itself directly or indirectly to solve a problem. Recursive solutions are often cleaner and more elegant for problems that have a naturally recursive structure.

Example:

```c
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

Problems:

1. Write a recursive function to calculate the sum of digits of a number.

2. Write a recursive function to reverse a string.

3. Write a recursive function to find the greatest common divisor (GCD) of two numbers.

4. Write a recursive function to solve the Tower of Hanoi problem.

---

2. Pointers

Pointers are variables that store the memory address of another variable. They are used for dynamic memory allocation, arrays, and functions.

Example:

```c
#include <stdio.h>

int main() {
    int x = 10;
    int *p = &x;

    printf("Value of x: %d\n", x);
    printf("Address of x: %p\n", p);
    printf("Value at address %p: %d\n", p, *p);

    return 0;
}
```

Problems:

1. Write a program to reverse an array using pointers.

2. Write a program to dynamically allocate memory for an array using pointers.

---

3. File Handling

File handling in C involves reading from and writing to files. It allows data to be stored permanently on a storage device.

Example:

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(file, "Hello, World!\n");
    fclose(file);

    return 0;
}
```

Problems:

1. Write a program to read a file and display its contents.

2. Write a program to count the number of lines in a file.

3. Write a program to append text to an existing file.

4. Write a program to copy the contents of one file to another.

5. Write a program to find and replace a word in a file.

---

4. Structures

Structures in C are user-defined data types that group related variables of different types.

Example:

```c
#include <stdio.h>

struct Student {
    char name[50];
    int age;
    float grade;
};

int main() {
    struct Student student1 = {"John Doe", 20, 85.5};

    printf("Name: %s\n", student1.name);
    printf("Age: %d\n", student1.age);
    printf("Grade: %.2f\n", student1.grade);

    return 0;
}
```

Problems:

1. Write a program to store and display information of 5 students using structures.

2. Write a program to calculate the average grade of students using structures.

---

5. Enums

Enums are user-defined data types that consist of integral constants. They help in writing more readable and maintainable code.

Example:

```c
#include <stdio.h>

enum Days {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};

int main() {
    enum Days today = WEDNESDAY;

    if (today == WEDNESDAY) {
        printf("Today is Wednesday.\n");
    }

    return 0;
}
```

Problems:

1. Write a program to demonstrate the use of enums for representing months of the year.

2. Write a program to use enums to represent traffic light signals and print the corresponding action.

3. Write a program to use enums to manage states in a simple finite state machine.

4. Write a program to use enums to categorize different types of books.

5. Write a program to use enums to represent different modes of transportation.

---

6. malloc:

malloc` is used for dynamic memory allocation in C. It allocates memory blocks at runtime and returns a pointer to the beginning of the block.

Example:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = (int*) malloc(sizeof(int) * 5);

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        ptr[i] = i + 1;
        printf("%d ", ptr[i]);
    }

    free(ptr);

    return 0;
}
```

Problems:

1. Write a program to dynamically allocate an array of integers using malloc.

2. Write a program to dynamically allocate memory for a 2D array using malloc.

3. Write a program to copy a string using malloc.

---

7. calloc

calloc` is used for dynamic memory allocation in C. It allocates memory blocks and initializes all bytes to zero.

Example:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = (int*) calloc(5, sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]); // All elements will be initialized to 0
    }

    free(ptr);

    return 0;
}
```

Problems:

1. Write a program to create a dynamic string array using calloc.

2. Write a program to store n floats using calloc.

---

8. Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1.

Example:

```c
#include <stdio.h>

void fibonacci(int n) {
    int t1 = 0, t2 = 1, nextTerm;
    for (int i = 1; i <= n; ++i) {
        printf("%d ", t1);
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;
    }
    printf("\n");
}

int main() {
    int n = 10;
    fibonacci(n);
    return 0;
}
```

Problems:

1. Write a program to generate the first n Fibonacci numbers.

2. Write a recursive function to find the nth Fibonacci number.

3. Write a program to check if a number is in the Fibonacci sequence.

4. Write a program to generate Fibonacci numbers up to a given number.

5. Write a program to print the Fibonacci series in reverse order.

---

9. Factorials

The factorial of a non-negative integer n is the product of all positive integers less than or equal to n. It is denoted by n!.

Example:

```c
#include <stdio.h>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    printf("Factorial of %d is %d\n", num, factorial(num));
    return 0;
}
```

Problems:

3. Write a program to find the factorial of a number using pointers.

4. Write a program to compute the factorial of an array of numbers.

5. Write a program to find the factorial of a number using dynamic programming.

==========

A D V A N C E P R O B L E M S

Pointers

1. Memory Management with Linked Lists:

Problem: Implement a doubly linked list with functions to insert nodes at the beginning, end, and middle. Additionally, create functions to delete nodes, reverse the list, and find a node by its value.

Challenge: Ensure proper memory management by preventing memory leaks and handling

edge cases such as empty lists and single-node lists.

---

2. Dynamic Multi-dimensional Arrays:

Problem: Write a program to dynamically allocate a 3D array using pointers. Implement functions to set and get values, and deallocate the memory properly.

Challenge: Handle varying dimensions and ensure that the memory is correctly freed without causing memory leaks or segmentation faults.

---

3. Custom Memory Allocator:

Problem: Implement a custom memory allocator that simulates the behavior of `malloc`, `calloc`, `realloc`, and `free`. Manage a large block of memory and handle allocations, deallocations, and resizing of memory blocks.

Challenge: Implement memory coalescing to merge free blocks and handle fragmentation.

---

4. Pointer-Based String Library:

Problem: Create your own string library with functions such as `strlen`, `strcpy`, `strcat`, `strcmp`, and `strstr` using pointers.

Challenge: Optimize the functions for efficiency and handle edge cases such as null pointers and overlapping memory regions in `strcpy` and `strcat`.

---

5. Sparse Matrix using Pointers:

Problem: Implement a sparse matrix using a linked list of linked lists (each row is a linked list

of non-zero elements). Provide functions for matrix addition, multiplication, and transpose.

Challenge: Ensure that the implementation efficiently handles insertion and retrieval of elements, and correctly performs matrix operations.

Recursion

1. Palindrome Partitioning:

Problem: Write a recursive function to partition a given string into all possible palindromic substrings.

Challenge: Optimize the function to avoid redundant checks and reduce the time complexity.

---

2. Rat in a Maze:

Problem: Given a maze represented by a 2D array, write a recursive function to find a path from the top-left corner to the bottom-right corner. The rat can move only right or down.

Challenge: Handle cases where no path exists and optimize to find the shortest path.

---

3. M-ary Tree Traversal:

Problem: Implement recursive functions to perform pre-order, post-order, and level-order traversals of an M-ary tree (a tree where each node can have at most M children).

Challenge: Manage the recursive stack to handle large trees efficiently.

---

4. Expression Evaluation:

Problem: Write a recursive function to evaluate a mathematical expression given as a string. The expression can contain integers, +, -, *, /, and parentheses.

Challenge: Handle operator precedence and parentheses correctly to ensure accurate evaluation.

---

5. Wildcard Pattern Matching:

Problem: Implement a recursive function to perform pattern matching with support for wildcard characters `*` (matches any sequence of characters) and `?` (matches any single character).

Challenge: Optimize the function to handle large input strings and patterns efficiently.

Strings

1. Longest Palindromic Substring:

Problem: Write a program to find the longest palindromic substring in a given string.

Challenge: Optimize the solution to run in O(n^2) time using dynamic programming or O(n) time using Manacher's algorithm.

---

2. Text Justification:

Problem: Write a function to justify a given text to a specified width. The function should distribute spaces between words evenly and handle edge cases like single words in a line.

Challenge: Ensure the solution is efficient and handles large inputs gracefully.

---

3. String Subsequence Matching:

Problem: Implement a function to find if a string `s1` is a subsequence of another string `s2`.

Challenge: Optimize the function to run in linear time.

---

4. Longest Repeated Substring:

Problem: Write a program to find the longest substring that appears at least twice in a given string.

Challenge: Use efficient algorithms like suffix arrays or suffix trees to achieve this.

---

5. Zigzag Conversion:

Problem: Convert a given string into a zigzag pattern on a given number of rows and then read it row by row. For example, the string "PAYPALISHIRING" on 3 rows would be written as:

```
P   A   H   N
A P L S I I G
Y   I   R
```

Challenge: Implement the conversion and reading process efficiently, handling edge cases where the number of rows is greater than the length of the string.

These problems will help deepen your understanding of pointers, recursion, and strings in C, and provide a good challenge to improve your problem-solving skills. Happy coding!

---()---