**Code relay**

1. **Challenge:**

Imagine you're a spy tasked with delivering classified information in the form of a sorted array. However, to avoid detection, you can't use traditional sorting algorithms. Instead, you have access to a "black box" function that takes two elements from the array and secretly compares them, returning:

* 1 if the first element is larger
* -1 if the first element is smaller
* 0 if they are equal

Your mission is to write a C program that utilizes this "black box" function to sort the entire array in ascending order without violating the secrecy rules. Remember, you only know the function's input/output behaviour, not its internal workings.

**Sample Input:**

Array: [7, 2, 4, 1, 9]

**Sample Output:**

Sorted Array: [1, 2, 4, 7, 9]

Solution:

#include <stdio.h>

// Function prototype for the "black box" comparison function

int compare(int a, int b);

// Function prototype for the partitioning step of QuickSort

int partition(int arr[], int low, int high);

// Function prototype for the QuickSort algorithm

void quickSort(int arr[], int low, int high);

// "Black box" comparison function

int compare(int a, int b) {

// Simulate comparison by using the given "black box" functio

1. **Problem: Minimum Cost**

You are given the prices of N items in a store. Your task is to determine whether you can purchase all the items within a budget of B rupees.

Each item has a specific price, and you can buy any quantity of each item (including zero). Determine whether it's possible to buy all the items without exceeding the budget.

**Input Format:**

The first line of input contains two space-separated integers N and B, where N represents the number of items in the store (1 ≤ N ≤ 100) and B represents the budget in rupees (1 ≤ B ≤ 10^5).

The second line contains N space-separated integers p1, p2, ..., pN, where pi represents the price of the i-th item (1 ≤ pi ≤ 1000).

**Output Format:**

Output "YES" if it's possible to buy all the items within the budget, otherwise output "NO".

You can print each character in uppercase or lowercase. For example, the strings "YES", "yes", "Yes", and "yES" are all considered identical.

**Sample Input:**

5 200

50 75 100 150 200

**Sample Output:**

Objective:

YES

**Explanation:**

In this case, the total cost of all items is 50 + 75 + 100 + 150 + 200 = 575, which is less than the budget of 200 rupees. Thus, it's possible to buy all the items within the budget.

**Solution:**

#include <stdio.h>

// Function to check if it's possible to purchase all items within the budget

int canPurchase(int prices[], int n, int budget) {

int total = 0;

// Calculate the total price of all items

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

total += prices[i];

}

// Check if the total price is within the budget

if (total <= budget) {

return 1; // It's possible to purchase all ite

1. Consider the task of swapping the values of three integer variables x, y, and z using pointers in a C program. However, here's the catch: you are not permitted to use the standard dereference operator (\*) or the address-of operator (&). Instead, devise a solution using pointer arithmetic and other creative techniques. The initial values of x, y, and z are 10, 20, and 30 respectively. Ensure that your solution effectively swaps these values without relying on conventional pointer operations.

**Input:**

* This challenge doesn't have a specific input. The initial values of three integer variables, x, y, and z, are already defined within the program as 10, 20, and 30, respectively.

**Output:**

The expected output is the swapped values of the three variables after the program execution. In this case, the output would be:

x = 20

y = 30

z = 10

**Solution:**

#include <stdio.h>

int main() {

int x = 10, y = 20, z = 30;

int \*ptr = &x;

// Swap x and y without using dereference operator

x = x + y;

y = x - y;

x = x - y;

// Move pointer to the next integer location (which is y)

ptr++;

// Swap y and z without using dereference operator

\*ptr = \*ptr + z;

z = \*ptr - z;

\*ptr = \*ptr - z;

// Move pointer to the previous integer location (which is x)

ptr--;

// Swap x and z wi

1. Two master code builders are having a friendly duel! One champions the elegance of recursion, while the other advocates for the efficiency of iteration. Can you create their weapons: functions to calculate the factorial of a number, each designed in their preferred style? Remember, the victor will be determined by both correctness and performance. Go forth and code, valiant knight of algorithms! So find factorial given number with efficient way.

**Sample input:**

**enter the number to find factorial: 5**

**SAMPLE Output:**

**factorial=120**

**Solution:**

**#include <stdio.h>**

**// Recursive function to calculate factorial**

**unsigned long long factorial(int n) {**

**// Base case: factorial of 0 is 1**

**if (n == 0) {**

**return 1;**

**}**

**// Recursive case: factorial of n is n times factorial of (n - 1)**

**return n \* factorial(n - 1);**

**}**

**int main() {**

**int num;**

**printf("Enter a non-negative integer: ");**

**scanf("%d", &num);**

**// Check if the input is non-negative**

**if (num < 0) {**

**printf("Factorial is not defined for negative numbers.\n");**

**} else {**

**unsigned long long fact = factorial(num);**

**printf("Factorial of %d is %llu\n", num, fact);**

**}**

**return 0;**

**}**

1. Imagine you're a rogue developer tasked with creating a custom memory allocator for a resource-constrained embedded system. This system has a fixed amount of memory, and your allocator needs to be extremely efficient in terms of both speed and memory usage. However, there's a twist: memory allocations can be of varying sizes, and you need to maximize space utilization to avoid fragmenting the memory pool.

Requirements::

* implement malloc and free functions that mimic the standard library behaviour.
* Utilize a fixed memory pool.
* Manage allocations of various sizes efficiently.

**Input:**

* You interact with the memory allocator through the standard malloc and free functions.
* malloc(size): Takes a single integer argument, the size (in bytes) of memory to allocate.
* free(ptr): Takes a single pointer argument, pointing to previously allocated memory to be released.

**Output:**

* malloc returns a pointer to the allocated memory block (or NULL if unsuccessful).
* free doesn't explicitly return anything, but the memory block becomes available for future allocations.

Solution:

#include <stdio.h>

#include <stdint.h>

// Define the maximum memory size for the allocator

#define MEMORY\_SIZE 1024 // Example size, adjust as needed

// Define the minimum block size (power of 2)

#define MIN\_BLOCK\_SIZE 8 // Example size, adjust as needed

// Define a structure to represent memory blocks

typedef struct Block {

uint8\_t \*start; // Start address of the block

uint32\_t size; // Size of the block

struct Block \*next; // Pointer to the next block

} Block;

// Define global variables for memory pool and free list

uint8\_t memory[MEMORY\_SIZE];

Block \*free\_list;

// Function to initialize the memory allocator

void init\_allocator() {

free\_list = (Block\*)memory;

free\_list->start = memory;

free\_list->size = MEMORY\_SIZE;

free\_list->next = NULL;

}

**6.)**

**CHALLENGE:** IMAGINE YOU'RE A TIME TRAVELLER TRAPPED IN A FORGOTTEN CALENDAR SYSTEM FROM A DISTANT CIVILIZATION. THEIR LEAP YEAR RULES ARE CRYPTIC:

• YEARS DIVISIBLE BY 3 ARE ALWAYS LEAP YEARS (STANDARD RULE).

• BUT WAIT, YEARS THAT ARE MULTIPLES OF 13 ALSO CANCEL OUT THE LEAP YEAR STATUS, EXCEPT FOR THOSE PERFECTLY DIVISIBLE BY 39!

• AND TO ADD ANOTHER LAYER OF CONFUSION, CERTAIN "FORBIDDEN YEARS" ARE DECLARED LEAP YEARS DESPITE NOT FOLLOWING THE PREVIOUS RULES. THESE FORBIDDEN YEARS ARE REVEALED AS THE PROGRAM RUNS (THINK OF THEM AS SECRET CODES FROM THE PAST).

CAN YOU CRACK THIS BIZARRE CALENDAR SYSTEM AND WRITE A C PROGRAM THAT ACCURATELY DETERMINES WHETHER A GIVEN YEAR IS A LEAP YEAR IN THIS CIVILIZATION, TAKING INTO ACCOUNT ALL THE STRANGE RULES AND HIDDEN EXCEPTIONS?

Sample Input:

Year: 2000 Forbidden years: [1900, 2100]

Sample Output:

Year 2000 is a leap year.

**Solution:**

#include <stdio.h>

int main() {

int year;

// Input the year from the user

printf("Enter the year: ");

scanf("%d", &year);

// Check if the year is divisible by 3 (standard rule)

if (year % 3 == 0) {

// Check if the year is a multiple of 13

if (year % 13 == 0) {

// Check if the year is perfectly divisible by 39

if (year % 39 == 0) {

printf("%d is a leap year in the bizarre calendar system.\n", year);

} else {

printf("%d is not a leap year in the bizarre calendar system.\n", year);

}

} else {

printf("%d is a leap year in the bizarre calendar system.\n", year);

}

} else {

printf("%d is not a leap year in the bizarre calendar system.\n", year);

}

return 0;

}

**7)** **Challenge☹(hard) :**

Your company, with 20 Sales and 15 Marketing employees, has decided to expand its network to include three additional departments: Engineering (30 employees), Finance (25 employees), and Customer Service (40 employees). Each department requires access to the shared drive, and the cost per employee connection remains the same ($50 for Sales, $40 for Marketing, and variable costs for the new departments). However, there are two additional constraints:

1. Budget limit: You have a maximum budget of $5000 for expanding the network.

2. Scalability: The network design should be scalable to accommodate potential future growth (up to 20% increase in each department).

**Input Format:**

**The input will** be provided in two lines:

Line 1: [Department 1 employees] [Department 2 employees] [Department 3 employees] [Department 4 employees] [Department 5 employees]

• Departments can be in any order (Sales, Marketing, Finance, Engineering, Customer Service).

• Replace department names with their corresponding employee count (e.g., 20 for Sales, 15 for Marketing).

Line 2: [Department 1 connection cost] [Department 2 connection cost] [Department 3 connection cost] [Department 4 connection cost] [Department 5 connection cost]

• Costs are integers representing the price per employee connection for each department.

• Separate costs with spaces.

**Output Format:**

Print a single line containing the following information:

• "Network Design:" Followed by a brief description of your proposed network topology (e.g., star, mesh).

• "Total Cost:" Followed by the calculated total cost of setting up the network (integer).

• "Scalability:" Followed by a statement indicating whether the design can accommodate future growth (e.g., "Scalable" or "Not Scalable").

• "Justification:" Briefly explain your design choices and why they meet the budget and scalability constraints.

**Sample Input:**

30 40 20 15 25 50 40 35 60 45

**Sample Output:**

Network Design: Star topology with central switch. Total Cost: 4450 Scalability: Scalable up to 20% growth in each department. Justification: Star topology minimizes cabling cost and allows easy expansion by adding switches for new departments. The total cost stays within the $5000 budget even with 20% growth (4450 \* 1.2 = 5340, still less than $5000).

**Solution:**

#include <stdio.h>

// Structure to represent department details

struct Department {

char name[20];

int employees;

float cost\_per\_employee;

};

int main() {

// Existing departments

struct Department existing\_departments[] = {

{"Sales", 20, 50.0},

{"Marketing", 15, 40.0}

};

int num\_existing\_departments = sizeof(existing\_departments) / sizeof(existing\_departments[0]);

// New departments

struct Department new\_departments[] = {

{"Engineering", 30, 0.0},

{"Finance", 25, 0.0},

{"Customer Service", 40, 0.0}

};

int num\_new\_departments = sizeof(new\_departments) / sizeof(new\_departments[0]);

// Maximum budget for expansion

float max\_budget = 5000.0;

// Calculate total cost for existing depa

**8) challenge:**

consider how you might approach this without explicitly checking each character for being a vowel. Can you think of a different way to solve this problem that doesn't involve checking each character against a list of vowels?

Input:

• A string of characters entered by the user.

Output:

• The count of non-vowel characters in the string.

For example:

**Input:**

**Enter a string : Hello word**

**Output:**

Number of non-vowels character are : 6

And there are : ('H', 'l', 'l', 'W', 'r', 'l').

**Solution:**

#include <stdio.h>

int main() {

char input[100];

printf("Enter a number: ");

scanf("%s", input);

int i = 0;

while(input[i] != '\0') {

i++;

}

// Checking if the last digit's ASCII value is even or odd

if ((input[i-1] - '0') % 2 == 0) {

printf("The last digit of the number is even.\n");

} else {

printf("The last digit of the number is odd.\n");

}

return 0;

}

**9.)**

Challenge:

"Can you guess whether the last digit of a mystery number is odd or even? Remember, you won't see the number itself. Think about the patterns of digits and how they behave. Take your time and make your guess based on your intuition. Ready to take on the challenge?"

Solution;

#include <stdio.h>

int main() {

int num;

printf("Enter a number: ");

scanf("%d", &num);

// Check if the last digit of the number is odd or even

int last\_digit = num % 10;

if (last\_digit % 2 == 0) {

printf("Is the last digit of the number odd or even? Answer: Even\n");

} else {

printf("Is the last digit of the number odd or even? Answer: Odd\n");

}

return 0;

}