1.Scalability Issue

```
#include<stdio.h>
#include<stdlib.h>
#define Max_users 500000

void check_users(int user){
    if(user>Max_users){
        printf("Platform Crashed! Too many current Users:%d \n",user);
        exit(1);
    }
    else{
        printf("Platform running smoothly with %d current users.\n",user);
    }


}
int main(){
    int cur_users;
    printf("Enter the Number of Users in the Amazon Platform:");
    scanf("%d",&cur_users);
    check_users(cur_users);
    return 0;


}
```

2. Recommendation Algorithm Failure

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

```
#include<stdlib.h>
#include<time.h>
#define product_recamendation 100
#define probabilty 0.02
int main(){
    int falied=0;
    srand(time(NULL));

    for(int i=0;i<product_recamendation;i++){
        double chance=(double)rand()/RAND_MAX;
        if(chance<probabilty){
            falied++;
        }
    }
    printf("Product Recommendations :%d\n",product_recamendation);
    printf("Failed Recommendations: %d\n",falied);
    return 0;
}
```

3. Inventory Optimization

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define Warehoues 10
#define Min 500
```

```
#define Max 2000
#define Max_product 100
#define Max_capcity 2000

void optimizeInventory(int product_no, int warehouse_cp[], int space[], int profit[]) {
    int dp[Warehoues + 1][Max_capcity + 1] = {0};

    // Iterate over each warehouse
    for (int w = 1; w <= Warehoues; w++) {
        int capacity = warehouse_cp[w - 1];

        for (int p = 0; p < product_no; p++) {
            for (int cap = capacity; cap >= space[p]; cap--) {
                if (cap >= space[p]) {
                    int newProfit = dp[w - 1][cap - space[p]] + profit[p];
                    if (newProfit > dp[w][cap]) {
                        dp[w][cap] = newProfit;
                    }
                }
            }
        }
    }

    int maxProfit = 0;
    for (int w = 1; w <= Warehoues; w++) {
        if (dp[w][warehouse_cp[w - 1]] > maxProfit) {
            maxProfit = dp[w][warehouse_cp[w - 1]];
        }
    }

    printf("\nMaximum Profit Achievable: %d\n", maxProfit);
}

int main() {
    int warehouse_cp[Warehoues];
    srand(time(NULL));

    printf("Warehouse Capacities:\n");
    for (int i = 0; i < Warehoues; i++) {
        warehouse_cp[i] = Min + rand() % (Max - Min + 1);
        printf("Warehouse-%d: %d\n", i + 1, warehouse_cp[i]);
    }

    int numProducts;

    printf("\nEnter number of product types: ");
    scanf("%d", &numProducts);

    int space[Max_product], profit[Max_product];

    printf("\nEnter space required and profit per unit for each product:\n");
    for (int i = 0; i < numProducts; i++) {
        printf("Product %d (Space Profit): ", i + 1);
        scanf("%d %d", &space[i], &profit[i]);
    }
    optimizeInventory(numProducts, warehouse_cp, space, profit);

    return 0;
}
```

4. Logistics and Supply Chain Optimization

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define Warehouse 5
#define location 10

typedef struct Edge {
    int destination;
    int weight;
    struct Edge* next;
} Edge;

typedef struct Vertex {
    int id;
    Edge* edges;
} Vertex;



Edge* createEdge(int destination, int weight) {
    Edge* newEdge = (Edge*)malloc(sizeof(Edge));
    newEdge->destination = destination;
    newEdge->weight = weight;
    newEdge->next = NULL;
    return newEdge;
}


void addEdge(Vertex* vertex, int destination, int weight) {
    Edge* newEdge = createEdge(destination, weight);
    newEdge->next = vertex->edges;
    vertex->edges = newEdge;
}


// Dijkstra's algorithm
void dijkstra(Vertex* graph[], int source, int distances[]) {
    int visited[Warehouse + location];
    for (int i = 0; i < Warehouse + location; i++) {
        distances[i] = INT_MAX;
        visited[i] = 0;
    }

    distances[source] = 0;

    for (int count = 0; count < Warehouse + location - 1; count++) {
        int minDistance = INT_MAX, minIndex = -1;
        for (int v = 0; v < Warehouse + location; v++) {
            if (visited[v] == 0 && distances[v] <= minDistance) {
                minDistance = distances[v];
                minIndex = v;
            }
        }

        if (minIndex == -1) break;

        visited[minIndex] = 1;

        Vertex* u = graph[minIndex];

        Edge* currentEdge = u->edges;
        while (currentEdge != NULL) {
            int v = currentEdge->destination;
            int weight = currentEdge->weight;

            if (!visited[v] && distances[minIndex] != INT_MAX &&
                distances[minIndex] + weight < distances[v]) {
                distances[v] = distances[minIndex] + weight;
            }
            currentEdge = currentEdge->next;
        }
    }
}
```

```
int main() {
    Vertex* graph[Warehouse + location];


    for (int i = 0; i < Warehouse + location; i++) {
        graph[i] = (Vertex*)malloc(sizeof(Vertex));
        graph[i]->id = i;
        graph[i]->edges = NULL;
    }



    addEdge(graph[0], 5, 10);
    addEdge(graph[0], 6, 15);
    addEdge(graph[1], 7, 20);
    addEdge(graph[1], 8, 25);
    addEdge(graph[2], 9, 30);
    addEdge(graph[3], 5, 5);
    addEdge(graph[3], 7, 12);
    addEdge(graph[4], 8, 18);
    addEdge(graph[4], 9, 22);
    for (int sourceWarehouse = 0; sourceWarehouse < Warehouse; sourceWarehouse++) {
        int distances[Warehouse + location];
        dijkstra(graph, sourceWarehouse, distances);

        printf("Shortest paths from Warehouse %d:\n", sourceWarehouse);
        for (int destinationLocation = Warehouse; destinationLocation < Warehouse + location;
destinationLocation++) {
            if (distances[destinationLocation] == INT_MAX) {
                printf("  To Location %d: Not reachable\n", destinationLocation);
            } else {
                printf("  To Location %d: %d\n", destinationLocation, distances[destinationLocation]);
            }
        }
        printf("\n");
    }

    for (int i = 0; i < Warehouse + location; i++) {
        Edge* currentEdge = graph[i]->edges;
        while (currentEdge != NULL) {
            Edge* temp = currentEdge;
            currentEdge = currentEdge->next;
            free(temp);
        }
        free(graph[i]);
    }
    return 0;
}
```

5: Technical Debt Reduction

```
#include <stdio.h>
int main() {
    int total_lines = 1000000;
    double debt_per_line = 0.1;
    double reduction_rate = 0.02;
    double threshold = 0.01;
    double total_debt = total_lines * debt_per_line;
    int iterations = 0;

    printf("Starting Technical Debt Reduction...\n");
    printf("Initial Technical Debt: %.2f lines\n", total_debt);



    while (total_debt > total_lines * threshold) {
        total_debt -= total_debt * reduction_rate;
        iterations++;

        if (iterations % 10 == 0) {
            printf("After %d iterations, remaining debt: %.2f lines\n", iterations, total_debt);
        }
    }
    printf("\nTechnical debt reduced below %.0f%% in %d iterations!\n", threshold * 100, iterations);
    return 0;
}
```

6.Order Fulfillment Optimization

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include<unistd.h>
#define STAGES 5

typedef struct {
    char *name;
    int time_required;
} Stage;

int compare(const void *a, const void *b) {
    return ((Stage *)a)->time_required - ((Stage *)b)->time_required;
}
int main() {
    srand(time(NULL));
Stage stages[STAGES] = {
    {"Order Receipt", rand() % 5 + 1},
    {"Inventory Allocation", rand() % 5 + 1},
    {"Packaging", rand() % 5 + 1},
    {"Shipping", rand() % 5 + 1},
    {"Delivery Confirmation", rand() % 5 + 1}
};

printf("Initial Order Processing Stages:\n");
for (int i = 0; i < STAGES; i++) {
    printf("%s - Time Required: %d sec\n", stages[i].name, stages[i].time_required);
}

qsort(stages, STAGES, sizeof(Stage), compare);

printf("\nOptimized Order Processing Stages:\n");
for (int i = 0; i < STAGES; i++) {
    printf("%s - Time Required: %d sec\n", stages[i].name, stages[i].time_required);
}
printf("\nSimulating Order Fulfillment:\n");
int total_time = 0;
for (int i = 0; i < STAGES; i++) {
    printf("Processing: %s... (%d sec)\n", stages[i].name, stages[i].time_required);
    total_time += stages[i].time_required;
    sleep(stages[i].time_required);
}
printf("\nOrder Fulfilled in %d seconds!\n", total_time);
return 0;
}
```