**1. Linear Search Algorithm for a Robot's Pathfinding System**

**Context**: Imagine a robot navigating a grid of cells, where each cell contains a unique integer identifier. The robot needs to find a specific cell with a target identifier. A linear search algorithm can be used for this task.

**Algorithm in C**:

#include <stdio.h>

int linearSearch(int grid[], int size, int target) {

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

if (grid[i] == target) {

return i; // Return the index of the target

}

}

return -1; // Return -1 if the target is not found

}

int main() {

int grid[] = {12, 3, 5, 7, 19, 6};

int size = sizeof(grid) / sizeof(grid[0]);

int target = 19;

int result = linearSearch(grid, size, target);

if (result != -1) {

printf("Target found at index: %d\n", result);

} else {

printf("Target not found.\n");

}

return 0;

}

**Time Complexity**:

* Best Case: O(1) (target is the first element)
* Worst Case: O(n) (target is the last element or not present)
* Average Case: O(n)

**Space Complexity**:

* Space Complexity: O(1)O(1)O(1) (in-place search, no additional memory required)

**Explanation**: This algorithm simulates a robot scanning a grid sequentially to locate a target identifier. The linear search is efficient when the grid is small or the target is near the start of the array.

**2. Bubble Sort Algorithm for Prioritizing Tasks in an Automated System**

**Context**: In an automation system, tasks need to be executed based on their priority. Bubble sort can be used to sort tasks based on their priority levels, with the highest priority tasks being executed first.

**Algorithm in C**:

#include <stdio.h>

void bubbleSort(int tasks[], int size) {

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (tasks[j] > tasks[j + 1]) {

// Swap tasks[j] and tasks[j+1]

int temp = tasks[j];

tasks[j] = tasks[j + 1];

tasks[j + 1] = temp;

}

}

}

}

int main() {

int tasks[] = {3, 1, 4, 1, 5, 9, 2};

int size = sizeof(tasks) / sizeof(tasks[0]);

bubbleSort(tasks, size);

printf("Tasks sorted by priority: ");

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

printf("%d ", tasks[i]);

}

printf("\n");

return 0;

}

**Time Complexity**:

* Best Case: O(n) (already sorted)
* Worst Case: O(n^2)(reverse sorted)
* Average Case: O(n^2)

**Space Complexity**:

* Space Complexity: O(1) (in-place sort, no additional memory required)

**Explanation**: The bubble sort algorithm is ideal for small sets of tasks where priorities change frequently. Although it's not the most efficient sorting algorithm for large datasets, its simplicity makes it suitable for real-time systems where ease of implementation is crucial.

**3. Max-Min Algorithm for Game AI Decision Making**

**Context**: In a game, the AI needs to evaluate the best possible move by considering both the maximum advantage and minimum risk. The Max-Min algorithm can be used to simulate this decision-making process.

**Algorithm in C**:

#include <stdio.h>

void findMaxMin(int arr[], int size, int \*max, int \*min) {

\*max = arr[0];

\*min = arr[0];

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

if (arr[i] > \*max) {

\*max = arr[i];

}

if (arr[i] < \*min) {

\*min = arr[i];

}

}

}

int main() {

int scores[] = {23, 89, 15, 98, 67, 45};

int size = sizeof(scores) / sizeof(scores[0]);

int max, min;

findMaxMin(scores, size, &max, &min);

printf("Max score: %d\n", max);

printf("Min score: %d\n", min);

return 0;

}

**Time Complexity**:

* Best Case: O(n)
* Worst Case: O(n)
* Average Case: O(n)

**Space Complexity**:

* Space Complexity: O(1) (constant extra space for max and min)

**Explanation**: This algorithm evaluates the best and worst outcomes (max and min scores) from a set of possible moves. This is critical in AI decision-making for games, where the AI must assess the best move by maximizing its gain while minimizing the opponent's advantage.