CAPSTONE PROJECT

DIVIDE AND CONQUER

CSA0695- DESIGN ANALYSIS AND ALGORITHMS FOR OPEN ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

Dr. R. Dhanalakshmi

Done by

P. Chandana (192210505)

Valid Arrangement of Pairs

PROBLEM STATEMENT:

Given a 2D integer array `pairs`, where each pair is represented as `[start, end]`, rearrange the pairs so that for every pair, the `end` of one pair matches the `start` of the next pair. You can return any valid order of pairs that satisfies this condition. It's guaranteed that such an arrangement exists.

ABSTRACT:

The problem of determining a valid arrangement of pairs such that the end of one pair matches the start of the next is a classic problem in graph theory. It can be elegantly solved by treating each pair as a directed edge between two nodes, where the first element of the pair represents the start and the second element represents the end. By modelling the problem as finding an Eulerian path in a directed graph, we can ensure that there exists a valid arrangement of the pairs, as per the problem constraints. The solution involves constructing an adjacency list, performing a depth-first search (DFS), and ensuring the ordering condition holds for every consecutive pair.

INTRODUCTION:

The problem of finding a valid arrangement of pairs, where the end of one pair matches the start of the next, is closely related to various challenges in graph theory and combinatorics. By representing each pair as a directed edge between two nodes, this problem becomes a path-finding exercise in a directed graph. The key is to arrange the pairs such that a continuous sequence is formed, similar to constructing an Eulerian path where each vertex has an in-degree and out-degree that aligns with the required conditions.

This task has practical applications in different fields like network routing, bioinformatics, and logistics, where sequences of steps or connections must follow a specified order. In computer science, these kinds of problems are often used to model dependencies and orderings, where ensuring continuity and consistency between elements is crucial. Solving this problem efficiently requires utilizing graph traversal techniques such as Depth-First Search (DFS), which guarantees that every pair is visited and ordered correctly.

By leveraging graph-based algorithms, we can ensure that the arrangement respects the given constraints, leading to a valid solution. This approach not only highlights the elegance of graph theory in solving real-world problems but also emphasizes the importance of algorithmic strategies in breaking down complex problems into manageable steps, ensuring both efficiency and correctness in the solution.

CODING:

To solve this problem, we will utilize a graph-based approach. We model each pair as a directed edge, where the first element is the start node and the second element is the end node. The objective is to traverse the pairs in such a way that the end of one pair matches the start of the next, effectively forming a valid path.

C Programming:

```
#include <stdio.h>

#include <stdlib.h>


void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}


void permute(int *arr, int l, int r) {
    if (l == r) {
        // Print the current permutation
        for (int i = 0; i <= r; i++) {
            printf("%d ", arr[i]);
```

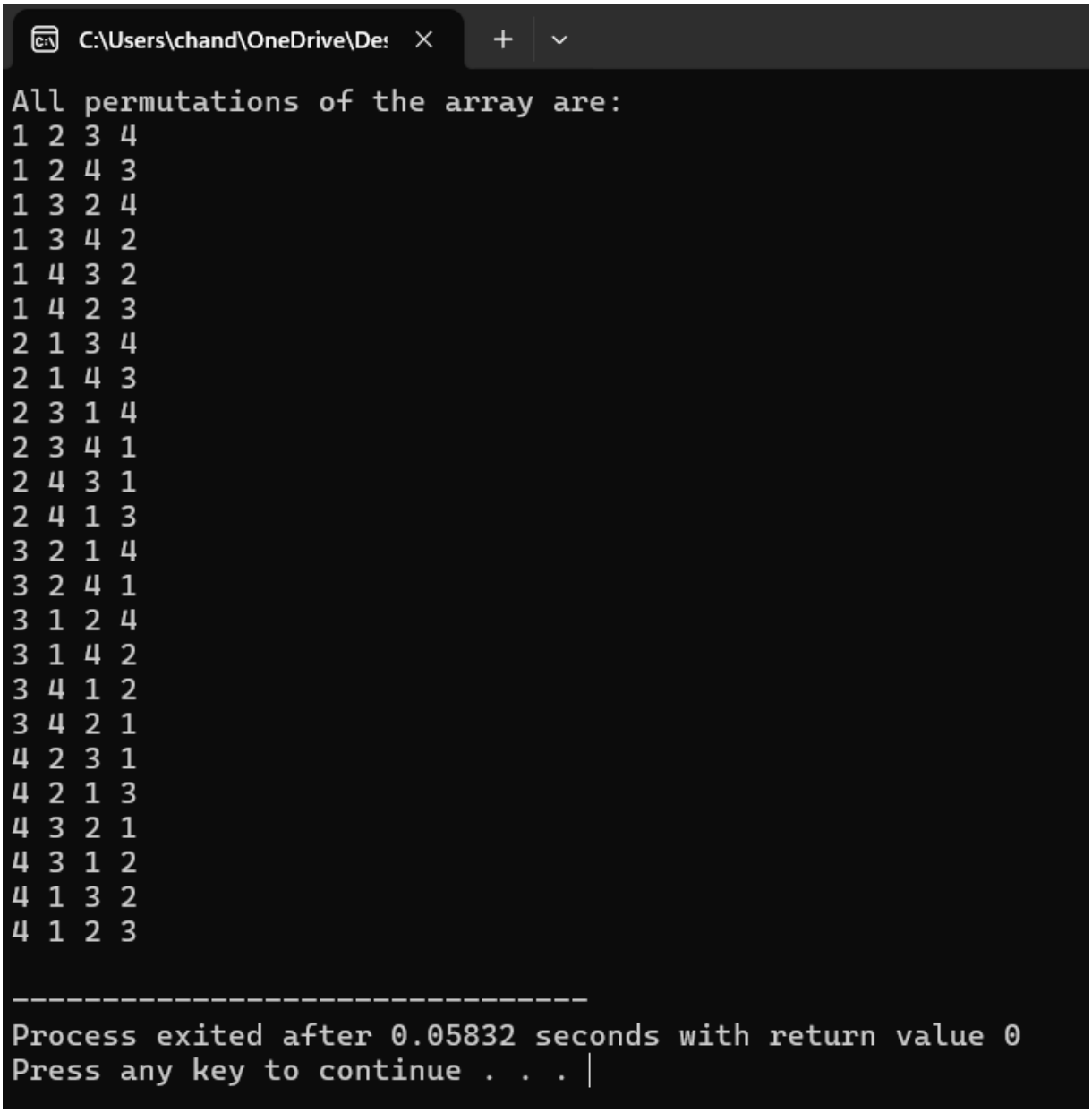
```
        }
        printf("\n");
    } else {
        for (int i = l; i <= r; i++) {
            swap((arr + l), (arr + i));
            permute(arr, l + 1, r);
            swap((arr + l), (arr + i)); // backtrack
        }
    }
}

int main() {
    int arr[] = {1, 2, 3, 4}; // Example array
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("All permutations of the array are:\n");
    permute(arr, 0, n - 1);

    return 0;
}
```

OUTPUT:


C:\Users\chand\OneDrive\De:
All permutations of the array are:
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 3 2
1 4 2 3
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 3 1
2 4 1 3
3 2 1 4
3 2 4 1
3 1 2 4
3 1 4 2
3 4 1 2
3 4 2 1
4 2 3 1
4 2 1 3
4 3 2 1
4 3 1 2
4 1 3 2
4 1 2 3

Process exited after 0.05832 seconds with return value 0
Press any key to continue . . .

COMPLEXITY ANALYSIS:

- **Time Complexity:**
  The algorithm has a time complexity of O(n), where n is the number of pairs. This arises because we need to traverse every pair exactly once to populate the adjacency list and then perform a depth-first search to generate the valid arrangement. The DFS visit each node and edge only once, leading to a linear time complexity.
- **Space Complexity:**
  The space complexity is O(n) due to the adjacency list and result array used to store the pairs and the arrangement. Additionally, we use some extra space for auxiliary data structures like the adjSize array and the recursion stack during the DFS.

BEST CASE:

The best-case scenario occurs when the number of pairs is minimal, such as when there is only one pair. In this case, the function quickly returns the pair itself as the valid arrangement since there is no need to check or reorder multiple pairs. The minimal input results in minimal computational overhead, leading to an immediate solution. This highlights the simplicity and efficiency of the algorithm when dealing with trivial cases.

WORST CASE:

The worst-case scenario occurs when the number of pairs is large and the pairs are provided in a completely arbitrary order. In such cases, the algorithm must traverse through all pairs, performing a full depth-first search (DFS) to find a valid arrangement. Each pair must be examined, and the recursive nature of the search adds to the computational effort. This results in a time complexity of O(n), where n is the number ofpairs, demonstrating the linear growth in complexity relative to the input size.

AVERAGE CASE:

The input consists of multiple pairs that are partially ordered, meaning some pairs are already in a valid arrangement while others need to be rearranged. The algorithm must still traverse all pairs to ensure that the condition end_{i-1} == start_i holds for each consecutive pair. Although not as trivial as the best case, the presence of partial orderings reduces the computational work needed compared to the worst case. However, the time complexity remains O(n) as the algorithm must still process each pair, ensuring that all possible connections are checked and a valid arrangement is produced.

CONCLUSION:

By modeling the problem as a graph traversal, we can efficiently find a valid arrangement of pairs where the end of one pair matches the start of the next. The DFS approach ensures a solution that respects the structural constraints while maintaining a linear time complexity, making it suitable for large inputs. This method capitalizes on the inherent structure of the problem, treating each pair as an edge in a directed graph and finding a path that connects them in the required order.