CSE 321 - Homework 3

Due date: 8/12/2022, 23:59

Çağla Şahin
Instructor's Name: Didem Gözüpek

1-b.Sol: Construct a non-DFS-based algorithm.

Here is a non-DFS based algorithm to obtain a topological ordering for a given DAG:

1. Create a list of nodes with in-degree 0, called the sources list.
2. Initialize an empty list for the topological ordering, called the ordering list.
3. While the sources list is not empty:
4.
    1. Remove a node from the sources list and append it to the ordering list.
    2. Decrement the in-degree of each of the node's children by 1.
    3. If a child's in-degree becomes 0, add it to the sources list.
5. If the ordering list contains all nodes in the DAG, return the ordering list as the topological ordering of the DAG. Otherwise, the DAG contains a cycle and no topological ordering exists.

2.Sol: Power algorithm which has O(logn) time complexity:

In this algorithm, the property that a^n = (a^2)^(n/2) is used to reduce the value of n by half in each iteration. This means that the number of iterations is logarithmic in n, giving us a worst-case time complexity of O(log n).

3.Sol: Sudoku solving with exhaustive search:

In this algorithm, recursive depth-first search is used to try all possible values for each empty grid in the Sudoku map. is_valid() function is used to check if a given value is valid for a given cell, and the get_box() function is used to retrieve the values of the cells in the same 3x3 box as a given cell. If the puzzle cannot be solved with the current values, it is restored with the original values and tried the next possible value.

4. Sol:

1 - Insertion Sort (given array is {6, 8, 9 ,8, 3, 3, 12})

Sorting steps of the given array using insertion sort:

1. Begin with the first element of the array, 6. Since it is the first element, it is already in its correct position.
2. Move to the next element, 8. Since 8 is already in its correct position, we skip it. The array remains the same, [6, 8, 9, 8, 3, 3, 12].
3. Move to the next element, 9. Since 9 is greater than 8, it is in its correct position. We skip it. The array is still [6, 8, 9, 8, 3, 3, 12].
4. Move to the next element, 8. Since 8 is already in the array, we skip it. The array remains the same, [6, 8, 9, 8, 3, 3, 12].
5. Move to the next element, 3. Since 3 is less than 6, it is not in its correct position. To do this, we shift all the elements greater than 3 to the right by one position, and then insert 3 in the empty space that is created. The array now becomes [3, 6, 8, 9, 8, 3, 12].
6. Move to the next element, 3. Since 3 is already in the array, we skip it. The array remains the same, [3, 6, 8, 9, 8, 3, 12].
7. Finally, move to the last element, 12. Since 12 is already in its correct position, we skip it. The array remains the same, [3, 6, 8, 9, 8, 3, 12].

After these steps, the array is sorted in ascending order. The final sorted array is **[3, 3, 6, 8, 8, 9, 12].**

A sorting algorithm is considered stable if it maintains the relative order of the elements with the same value. In other words, if two elements have the same value, they should maintain the same order after the sorting is performed. In the case of the given array, both 3's and 8's were placed in their correct positions and remained next to each other in the sorted array. This indicates that the insertion sort algorithm is **stable.**

2 - Quick Sort (given array is {6, 8, 9 ,8, 3, 3, 12})

Sorting steps of the given array using quick sort:

1. Select a pivot element from the array. This can be any element, but for simplicity, let's choose the first element, 6, as the pivot.
2. Divide the array into two sub-arrays: one containing all elements less than or equal to the pivot, and the other containing all elements greater than the pivot. In this case, the two sub-arrays would be {3, 3} and {8, 8, 9, 12}.
3. Sort each of the two sub-arrays using the quick sort algorithm.
4. Finally, combine the two sorted sub-arrays and the pivot element to get the final sorted array: **{3, 3, 6, 8, 8, 9, 12}.**

In the given array, the two elements with the value of 8 are in the positions 2 and 4. After sorting the array using the quick sort algorithm, these two elements will be in the positions 4 and 5. This means that the quick sort algorithm does not preserve the relative order of elements with the same value, and therefore **it is not a stable sorting algorithm.**

2 - Bubble Sort (given array is {6, 8, 9 ,8, 3, 3, 12})

Sorting steps of the given array using bubble sort:

6. First, compare first 2 elements; 6 and 8. 6 is less than 8, so they will stay in same places.
7. 8<9, stay same.
8. 9>8, swap. ({6, 8, 8, 9, 3, 3, 12})
9. 9>3, swap. ({6, 8, 8, 3, 9, 3, 12})
10. 9>3, swap. ({6, 8, 8, 3, 3, 9, 12})
11. 9<12, stay same, end of array. Going back to head of array for check again till there is no swap round.
12. Array is now: ({6, 8, 8, 3, 3, 9, 12})

13. 6<8. stay same.
14. 8<=8, stay same.
15. 8>3, swap. ({6, 8, 3, 8, 3, 9, 12})
16. 8>3, swap. ({6, 8, 3, 3, 8, 9, 12})
17. Till 12, all ascending - no swap. (checked with bubble sort, I didn't note). Back to head.
18. Array is now: ({6, 8, 3, 3, 8, 9, 12})
19. 6<8. stay same.
20. 8>3, swap. ({6, 3, 8, 3, 8, 9, 12})
21. 8>3, swap. ({6, 3, 3, 8, 8, 9, 12})
22. Till 12, all ascending - no swap. Back to head.
23. Array is now: ({6, 3, 3, 8, 8, 9, 12})
24. 6>3, swap. ({3, 6, 3, 8, 8, 9, 12})
25. 6>3, swap. ({3, 3, 6, 8, 8, 9, 12})
26. Now it is all ascending. The final sorted array is ({3, 3, 6, 8, 8, 9, 12})

The bubble sort algorithm is a **stable** sorting algorithm. This means that if the two items are considered equal with respect to the sort key, their original order will be preserved in the sorted output.

5. Sol:

5.1: Relation between brute force and exhaustive search

Brute force and exhaustive search are two algorithms for solving problems that trying every possible solution until the correct solution is found. Both of them involve systematically enumerating all possible solutions to a problem in order to find the one that works.

The main difference between the two methods is that brute force is a general approach to problem-solving that doesn't guarantee that the correct solution will be found, while exhaustive search guarantees that the correct solution will be found, provided that it exists.

In computer science area, brute force is often used as a way of solving problems when there is no more efficient method is known. This approach can be effective, but it can also be time-consuming and may not always produce the desired result.

Exhaustive search, on the other hand, is a more systematic and organized approach to problem-solving that guarantees that all possible solutions to a problem will be considered. This approach is typically more time-efficient than brute force, but it can still be impractical for problems with a large number of possible solutions.

To sum up, both brute force and exhaustive search are methods for solving problems by trying every possible solution, but exhaustive search is a more efficient and organized approach that guarantees that the correct solution will be found if it exists.

5.2: Caesar’s Cipher&AES and their vulnerabilities to brute force attacks.

Caesar's cipher is a encryption technique in which each letter in the plaintext is replaced with a letter that is a fixed number of positions down the alphabet. Caesar's cipher is not very secure, since there are only 26 possible shift values, so it can be **easily broken using a brute force attack.**

AES, on the other hand, is a more advanced form of encryption that uses a fixed block size and a key of varying length (128, 192, or 256 bits) to encrypt the plaintext. It is considered to be **very secure and is not vulnerable to brute force attacks**, because it uses complex mathematical operations.

It can be said that Caesar's cipher is a simple and relatively insecure form of encryption, while AES is a more advanced and secure form of encryption that is not vulnerable to brute force attacks.

5.3: The reason behind of growing exponentially of naive solution to primality testing.

A primality test is an algorithm for determining whether an input number is prime. The naive solution to primality testing grows exponentially because **it involves checking each number from 2 to n-1** to see if it divides n. Since there are n-2 numbers to check, the number of operations required grows exponentially as n increases.

For example, if n=6, then the naive algorithm would need to check 2, 3, 4 and 5 to see if any of them divide 6. If n=10, then it would need to check 2, 3, 4, 5, 6, 7, 8 and 9. If n=100, then it would need to check 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, and so on, up to 99. It can be observed that the number of operations required increases rapidly as n increases.