CSE 321 INTRODUCTION TO ALGORITHM DESIGN

HOMEWORK 3

Burcu Sultan ORHAN

1901042667

QUESTION 1:

In this implementation, I used a graph class and helper methods:

-addEdge: Adds an edge to graph.

-setVisited: Sets ‘visited’ info of edges.

-topologicalSortDFS: Sorts the graph topologically using depth first search with stack. It searches the graph in reverse order because it should reach the ‘depth’ first. Uses helper method to detect visited nodes.

-topologicalSortBFS: Sorts the graph topologically using breadth first search with queue and heap. It searches the graph nodes, puts them in queue, and Works.

WORST CASE TIME COMPLEXITY: O(n)

QUESTION 2:

In this question, I implemented a recursive method to solve exponantial function a^n. Base case is when n=1, it means a^1 which is a, so it returns just a. And when n is an even number, it returns a*findExponantial(a, n-1). This means basically a^n = a * a^(n-1). And finally, it calls findExponantial(a,n/2), and multiplies with itself so we get the result.

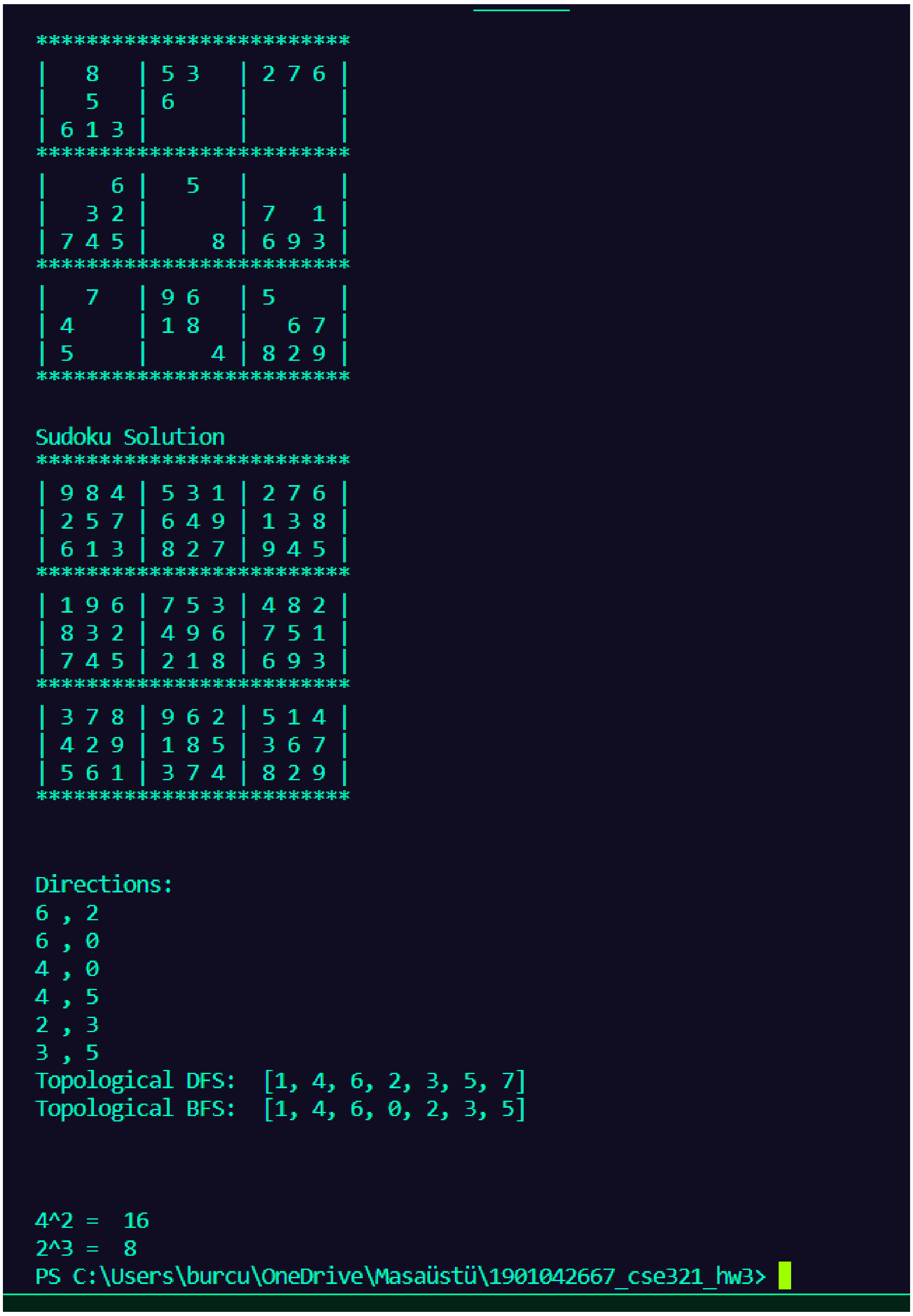
WORST CASE TIME COMPLEXITY: O(log(n))

QUESTION 3:

In this implementation, I used a helper function to draw sudoku puzzle. And firstly, program takes 9x9 array as input of sudoku. Then converts it into a string. Then, program finds every 0 value one by one. At each 0, it checks the other values on the same row, column and block as 0. It fills the ‘unusuble’ set with these existing values. Then, fills the ‘possibilities’ set with remaining values. Then draws the resulting sudoku.

WORST CASE TIME COMPLEXITY: O(9^n)

OUTPUTS:



```
*************************
|   8   | 5 3   | 2 7 6 |
|   5   | 6     |       |
| 6 1 3 |       |       |
*************************
|     6 |   5   |       |
|   3 2 |       | 7   1 |
| 7 4 5 |     8 | 6 9 3 |
*************************
|   7   | 9 6   | 5     |
| 4     | 1 8   |   6 7 |
| 5     |     4 | 8 2 9 |
*************************

Sudoku Solution
*************************
| 9 8 4 | 5 3 1 | 2 7 6 |
| 2 5 7 | 6 4 9 | 1 3 8 |
| 6 1 3 | 8 2 7 | 9 4 5 |
*************************
| 1 9 6 | 7 5 3 | 4 8 2 |
| 8 3 2 | 4 9 6 | 7 5 1 |
| 7 4 5 | 2 1 8 | 6 9 3 |
*************************
| 3 7 8 | 9 6 2 | 5 1 4 |
| 4 2 9 | 1 8 5 | 3 6 7 |
| 5 6 1 | 3 7 4 | 8 2 9 |
*************************


Directions:
6 , 2
6 , 0
4 , 0
4 , 5
2 , 3
3 , 5
Topological DFS:  [1, 4, 6, 2, 3, 5, 7]
Topological BFS:  [1, 4, 6, 0, 2, 3, 5]



4^2 =  16
2^3 =  8
PS C:\Users\burcu\OneDrive\Masaüstü\1901042667_cse321_hw3>
```

QUESTION 4:

$array$ = {6, 8, 9, 8, 3, 3, 12}

Since we want to see whether the sorting algorithm is stable or not, we will mark recurring elements like this:

$array$ = {6, 8′, 9, 8′′, 3′, 3′′, 12}

- Insertion Sort

  Step 1: Assume first element is already sorted, pick 8′ as key, compare key with sorted array. 6 < 8′, so not swapping.

  $array$ = {6, 8′, 9, 8′′, 3′, 3′′, 12}

  Step 2: Pick 9 as key, compare key with sorted array. 8′ < 9, so not swapping.

  $array$ = {6, 8′, 9, 8′′, 3′, 3′′, 12}

  Step 3: Pick 8′′ as key, compare key with sorted array. 9 > 8′′, so they get swept. 8′ = 8′′, so not swapping.

  $array$ = {6, 8′, 8′′, 9, 3′, 3′′, 12}

  Step 4: Pick 3′ as key, compare key with sorted array. 9 > 3′, so they get swept. 8′′ > 3′, so they get swept. 8′ > 3′, so they get swept. 6 > 3′, so they get swept.

  $array$ = {3′, 6, 8′, 8′′, 9, 3′′, 12}

  Step 5: Pick 3′′ as key, compare key with sorted array. 9 > 3′′, so they get swept. 8′′ > 3′′, so they get swept. 8′ > 3′′, so they get swept. 6 > 3′′, so they get swept. 3′ = 3′′, so not swapping.

  $array$ = {3′, 3′′, 6, 8′, 8′′, 9, 12}

Step 6: Pick 12 as key, compare key with sorted array. 9 < 12, so not swapping.

$array$ = {3’, 3’’, 6, 8’, 8’’, 9, 12}

We reached array’s length and we’re done with sorting. Array is now sorted. And as we can see, index of 3’ is smaller than the index of 3’’, just like in the beginning. Also the index of 8’ is smaller than the index of 8’’, just like in the beginning.

So, insertion sort is stable.

$array$ = {6, 8’, 9, 8’’, 3’, 3’’, 12}

- Quick Sort

  Step 1: Select 6 as pivot. Compare rightmost with pivot. 6 < 12, so not swapping. Shift from right, compare 3’’ with pivot. 6 > 3’’, so they get swept.

  $array$ = {3’’, 8’, 9, 8’’, 3’, 6 (pivot), 12}

  Step 2: Compare pivot from its left, 8’ > 6, so they get swept.

  $array$ = {3’’, 6 (pivot), 9, 8’’, 3’, 8’, 12}

  Step 3: Compare pivot from its right. 3’ < 6, so they get swept.

  $array$ = {3’’, 3’, 9, 8’’, 6 (pivot), 8’, 12}

  Step 4: Compare pivot from its left. 9 > 6, so they get swept.

$array$ = {3’’, 3’, 6 (pivot), 8’’, 9 , 8’, 12}

Step 5: So now, pivot’s right and left is sorted according to pivot. From now on, array gets partitioned with pivot’s right and left values seperately. Two halves get sorted. And final array is like this:

$array$ = {3’’, 3’, 6, 8’’, 8’, 9, 12}

We’re done with sorting. Array is now sorted. And as we can see, index of 3’ is bigger than the index of 3’’, unlike in the beginning. Also the index of 8’ is bigger than the index of 8’’, unlike in the beginning.

So, quick sort is NOT stable.

$array$ = {6, 8’, 9, 8’’, 3’, 3’’, 12}

- Bubble Sort

  Step 1: Take first 2 elements, 8’ > 6, so not swapping. 9 > 8’, so not swapping. 8’’ < 9, so they get swept. 3’ < 9, so they get swept. 3’’ < 9, so they get swept. 12 > 9, so not swapping.

  $array$ = {6, 8’, 8’’, 3’, 3’’, 9, 12}

  Step 2: Take first 2 elements, 8’ > 6, so not swapping. 8’’ = 8’, so not swapping. 3’ < 8’’, so they get swept. 3’’ < 8’’, so they get swept. 9 > 8’’, so not swapping.

  $array$ = {6, 8’, 3’, 3’’, 8’’, 9, 12}

  Step 3: Take first 2 elements, 8’ > 6, so not swapping. 3’ < 8’, so they get swept. 3’’ < 8’, so they get swept. 8’ = 8’’, so not swapping.

  $array$ = {6, 3’, 3’’, 8’, 8’’, 9, 12}

Step 4: Take first 2 elements, 3’ < 6, so they get swept. 3’’ < 6, so they get swept. 8’ > 6, so not swapping.

$array$ = {3’, 3’’, 6, 8’, 8’’, 9, 12}

Step 5: Take first 2 elements, 3’ = 3’’, so not swapping.

$array$ = {3’, 3’’, 6, 8’, 8’’, 9, 12}

We reached array’s length and we’re done with sorting. Array is now sorted. And as we can see, index of 3’ is smaller than the index of 3’’, just like in the beginning. Also the index of 8’ is smaller than the index of 8’’, just like in the beginning.

So, bubble sort is stable.

QUESTION 5:

a) A brute force algorithm solves a problem through exhaustion: it goes through all possible choices until a solution is found. The time complexity of a brute force algorithm is often proportional to the input size. Brute force algorithms are simple and consistent, but very slow.

   Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element. Note that although the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects.

b) The Caesar Cipher technique is one of the earliest and simplest methods of encryption technique. It’s simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet. For example with a shift of 1, A would be replaced by B, B would become C, and so on. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials. Thus to cipher a given text we need an integer value, known as a shift which indicates the number of positions each letter of the text has been moved down. encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25.

The AES algorithm (also known as the Rijndael algorithm) is a symmetrical block cipher algorithm that takes plain text in blocks of 128 bits and converts them to ciphertext using keys of 128, 192, and 256 bits. Since the AES algorithm is considered secure, it is in the worldwide standard. The AES algorithm uses a substitution-permutation, or SP network, with multiple rounds to produce ciphertext. The number of rounds depends on the key size being used. A 128-bit key size dictates ten rounds, a 192-bit key size dictates 12 rounds, and a 256-bit key size has 14 rounds. Each of these rounds requires a round key, but since only one key is inputted into the algorithm, this key needs to be expanded to get keys for each round, including round 0.

As we can see, AES is a much more complex and secure algorithm than Caesar’s Cipher. Also caesar’s cipher is very vulnerable to brute force attacks whereas AES is not very much. Caesar’s cipher is vulnerable because once you tried and found the shift number, which is a finite variable, you can decode entire ciphertext. AES is a more complex decryption algorithm and is not vulnerabl like Caesar’s cipher.

c) Assuming, that divisibility of two numbers is a unit operation (which is not correct for large numbers), algorithm's complexity is $O(\sqrt{n})$. The last improvement changes the constant only. When checking just prime divisors, algorithm's complexity becomes $O(\sqrt{n} / \ln(n))$.

   The main application of primes is cryptography, which expects us to generate primes with hundreds of digits. Apparently, the algorithm won't check numbers around 10100 in a reasonable time. Though, algorithm is applied on practice to do a "pre-check". A huge number being tested for primality is examined to be divisible by first million primes. If no divisors found, probabilistic algorithm is used then.