

GTU Department of Computer Engineering
CSE 321 - Fall 2022
Homework 3 Report

Emirkan Burak Yılmaz
1901042659

Q1. Topological Sorting

a. DFS Based Algorithm

```
13 def topologicalSortDFSHelper(self, v, visited, stack):
14     # Mark the current node as visited
15     visited[v] = True
16
17     # Recur for all the vertices adjacent to this vertex
18     for i in self.graph[v]:
19         if visited[i] == False:
20             self.topologicalSortDFSHelper(i, visited, stack)
21
22     # Push current vertex to result stack
23     stack.insert(0, v)
24
25
26 def topologicalSortDFS(self):
27     # Mark all the vertices as not visited
28     visited = [False] * self.numV
29     stack = [] # Stack to keep the topological order
30
31     # Get the topological order with DFS
32     for i in range(self.numV):
33         if visited[i] == False:
34             self.topologicalSortDFSHelper(i, visited, stack)
35
36     return stack
```

First create a size V (the number of vertices) array. Mark all the vertices as unvisited by setting all the values of array false. With this array the visit status of any vertex can be found mapping by the vertex id as array index. Also keep a stack to keep the topological order. After initialization process is done, topological sort is start by calling helper function for all the unvisited vertices. The helper function applies an algorithm based on DFS. First sets the current vertex as visited and then apply recursive calls for all the neighbour vertices. When there is no unvisited neighbour remain, the current vertex is pushed to the stack. After executing all the recursive calls, the stack contains the vertices in topological order.

Since the algorithm requires to traverse all the vertices in depth first search the running time same as DFS time complexity $O(V + E)$ where V is the number of vertices and E is the number of edges.

b. Non-DFS Based Algorithm

```
39 def topologicalSortNonDFS(self):
40     # Initilize an array to keep the inDegree's of vertices
41     inDegree = [0]*(self.numV)
42
43     # Traverse adjacency lists to fill indegrees of vertices
44     for i in range(self.numV):
45         for j in self.graph[i]:
46             inDegree[j] += 1
47
48     # Create an queue and enqueue all vertices with indegree 0
49     queue = []
50     for i in range(self.numV):
51         if inDegree[i] == 0:
52             queue.append(i)
53
54     topOrder = [] # Keeps the topological order
55
56     while queue:
57         u = queue.pop(0)
58         topOrder.append(u)
59
60         # Iterate through all the neighbour vertices
61         # and decrease their indegree by 1
62         for i in self.graph[u]:
63             inDegree[i] -= 1
64             if (inDegree[i] == 0):
65                 queue.append(i)
66
67     return topOrder if len(topOrder) == self.numV else None
```

The non-DFS version of the algorithm based on decrease-and-conquer technique. First keep an array to hold the indegree value of the vertices. Set all vertices as indegree 0 and traverse the adjacency list to find the indegree values for the vertices. Then put all the indegree 0 vertices into a queue. And finally keep a list to keep the topologically ordered vertices. After the initialization process is done, apply a loop till the queue becomes empty. Inside the loop enqueue a vertex from the queue and append it into the list which holds the topological order. Decrease indegree values of all the neighbour vertices by one and put the indegree 0 vertices into the queue. When the queue becomes empty the topological sort is figured out.

Let's say V is the number of vertices and E is the number of edges. The indegree value of each vertex needs to be found which takes $O(E)$ time. Dequeuing a vertex and decreasing indegree value of all the neighbour vertices and enqueueing the indegree 0 vertices takes $O(V + E)$ time. So, overall running time complexity of this algorithm is $O(V + E)$.

Q2. a^n (exponent)

```
3
4 def expHelper(y, a, n):
5     if n == 0:
6         return y
7     elif n % 2 == 0:
8         return expHelper(y, a * a, n / 2)
9     else:
10        return expHelper(a * y, a * a, (n - 1) / 2)
11
```

Naive solution of exponential expression requires $n-1$ multiplication operation, and it ends up $O(n)$ time complexity. However, the same result can be achieved with a smaller number of multiplication operation by squaring method. For instance, a^{21} can be expressed as

$$(((a^2)^2 \times a)^2)^2 \times a = (((a^4 \times a)^2)^2 \times a) = (((a^5)^2)^2 \times a) = ((a^{10})^2 \times a) = (a^{20} \times a) = a^{21}$$

By using squaring method in each recursive call, the least significant digits of the binary representation of n are removed. The number of recursive calls needs to made is $\text{ceil}(\log_2 n)$ which is equivalent to number of bits in the binary representation of n . So, time complexity of this algorithm is $O(\log n)$.

Q3. Sudoku

```
1 def isProper(board, size, row, col, num):
2     # check if num is used or not in the same row and column
3     for i in range(size):
4         if board[row][i] == num or board[i][col] == num:
5             return False
6     # check if num is used or not in the current 3x3 matrix
7     startRow = row - row % 3
8     startCol = col - col % 3
9
10    for i in range(3):
11        for j in range(3):
12            if board[startRow + i][startCol + j] == num:
13                return False
14
15    # it's proper to place num to board[row][col]
16    return True
17
18 def sudokuSolver(board, size):
19     return sudokuSolverHelper(board, size, 0, 0)
20
21 def sudokuSolverHelper(board, size, row, col):
22     # make sure not exceed the board boundaries
23     if col == size:
24         # check if the sudoku is solved
25         if row == size - 1:
26             return True
27         row += 1
28         col = 0
29
30     # if the current cell already contains a value, then continue with the next cell
31     if board[row][col] > 0:
32         return sudokuSolverHelper(board, size, row, col + 1)
33     for num in range(1, size + 1):
34         # check if num can be placed the current cell
35         if isProper(board, size, row, col, num):
36             # fill the cell with num and
37             board[row][col] = num
38             # continue recursion with next cell and make sure our assumption is correct
39             if sudokuSolverHelper(board, size, row, col + 1):
40                 return True
41             # the assumption was wrong so try another proper value
42             board[row][col] = 0
43     return False
44
```

Algorithm starts with the first cell of the board and continue to solution till the last cell. In each cell, a proper value from 1 to 9 is specified by considering that the selected value does not used in the current row and columns. After finding such a value, algorithm suggests assuming that selection is correct by filling the cell with that value. Then recursively continue with the next cell. If the game turns out to be unsolvable with this assumption, then it's clear that the made assumption was wrong. At this stage, if there are other possible values exist, make another assumption by selecting another value. However, if there is no possible value left, then return false to indicate the puzzle is not solvable with given initial cell assignments. If the game is designed to be solvable with the initial values, the algorithm tries several possible combinations and eventually solves the puzzle.

For every unassigned index there are 9 possible options so the upper bound time complexity of this algorithm $O(9^{(N^2)})$ for size $N \times N$ board.

Q4. Sorting

Array = {6, 8, 9, 8, 3, 3, 12}

a. Insertion Sort

6 8 9 8 3 3 12

6 8 9 8 3 3 12

6 8 9 8 3 3 12


6 8 8 9 3 3 12

3 6 8 8 9 3 12

3 3 6 8 8 9 12

3 3 6 8 8 9 12

b. Quick Sort

 : quick sort working range

 : partition result

6 8 9 8 3 3 12 (quicksort)

6 8 9 8 3 3 12 (partition)

6 8 9 8 3 3 12

6 3 9 8 3 8 12

6 3 3 8 9 8 12

3 3 6 8 9 8 12 (place the pivot)

3 3 6 8 9 8 12 (partitioned)

3 3 6 8 9 8 12 (quicksort)

3 3 6 8 9 8 12 (partition)

3 3 6 8 9 8 12 (place the pivot)

3 3 6 8 9 8 12 (partitioned)

3 3 6 8 9 8 12 (quicksort)

3 3 6 8 9 8 12 (quicksort)

3 3 6 8 9 8 12 (partition)
 3 3 6 8 9 8 12
 3 3 6 8 9 8 12 (place the pivot)
 3 3 6 8 9 8 12 (partitioned)
 3 3 6 8 9 8 12 (quicksort)
 3 3 6 8 9 8 12 (partition)
 3 3 6 8 9 8 12
 3 3 6 8 8 9 12 (place the pivot)
 3 3 6 8 8 9 12 (partitioned)
 3 3 6 8 8 9 12 (quicksort)
 3 3 6 8 8 9 12
 3 3 6 8 8 9 12 (quicksort)
 3 3 6 8 8 9 12

c. Bubble Sort

6 8 9 8 3 3 12
 6 8 9 8 3 3 12
 6 8 9 8 3 3 12
 6 8 8 9 3 3 12
 6 8 8 3 9 3 12
 6 8 8 3 3 9 12
 6 8 8 3 3 9 12
 6 8 8 3 3 9 12
 6 8 8 3 3 9 12
 6 8 3 8 3 9 12
 6 8 3 3 8 9 12
 6 8 3 3 8 9 12
 6 8 3 3 8 9 12

6 8 3 3 8 9 12
 6 3 8 3 8 9 12
 6 3 3 8 8 9 12
 6 3 3 8 8 9 12
 6 3 3 8 8 9 12
 3 6 3 8 8 9 12
 3 3 6 8 8 9 12
 3 3 6 8 8 9 12
 3 3 6 8 8 9 12
 3 3 6 8 8 9 12
 3 3 6 8 8 9 12
 3 3 6 8 8 9 12
 3 3 6 8 8 9 12

Are they stable?

Both bubble sort and insertion sort apply comparison to consecutive elements and in case of equality than the order of two consecutive elements does not change. For that reason, both bubble sort and insertion sort are stable sorting algorithms. On the other hand, quick sort is an unstable sorting algorithm because elements are swapped according to the pivot's position without considering their original position.

Q5. Quick Answer Questions

a. Brute Force and Exhaustive Search

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. A brute force approach involving search for a solution with a special property or constraint by exploring every possible combination from a set of choices or values. Exhaustive search is simply a brute force approach to combinational problems. It suggests generating each element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element.

b. Caesar's Cipher and AES Encryption

Caesar's cipher is a simple character substitution method. The cypher can be broken by brute force attack where the maximum number of combinations is only 26. The cracking can be done in a shorter time by analysing the frequency of letters and mapping them with the most common used English letters which are E, T and A etc.

AES relies on substitution-permutation network principle which means it is performed using a series of linked operations which involves replacing and shuffling of the input data. AES performs operations on bytes of data rather than its bits, so there are 16 bytes for 128 bit and 16 bytes can be consider as grid in a column major arrangement. 128-bit AES is safe against any brute force attacks. However, the key size used for encryption should always large enough that it could not be cracked by modern computer despite considering advancements in processor speeds based on Moore's law.

c. Naive Primality Test Exponential Running Time

The input size is typically measured in bits. To represent the number n , the input size should be $\log_2 n$. The naive primality test is linear in the value of input, but exponential in the size of the input. For instance, if the number n consisting of b bits, then $b = O(\log_2 n)$ and $n = O(2^b)$. So, the naive solution grows exponentially with the time complexity $O(2^b)$.