# CSE 321 HW3

Burak Kocausta

1901042605

**Menu Test for Q1, Q2, Q3:**

This menu will show up after built in test results.

This is for input = 1

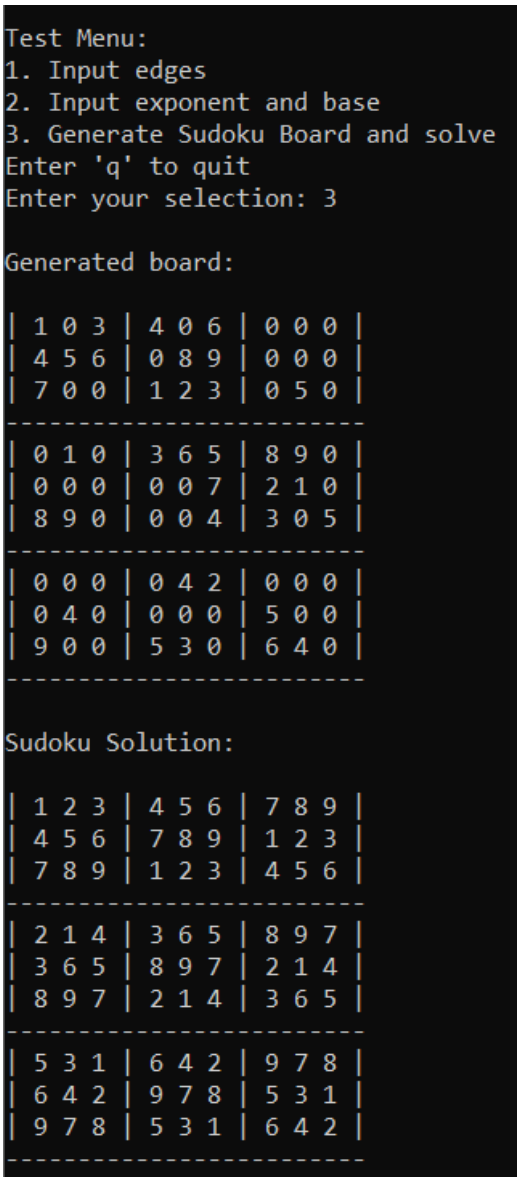
```
Test Menu:
1. Input edges
2. Input exponent and base
3. Generate Sudoku Board and solve
Enter 'q' to quit
Enter your selection: 1
(q for quit)edge source = CSE102
(q for quit)edge destination = CSE241
(q for quit)edge source = CSE241
(q for quit)edge destination = CSE222
(q for quit)edge source = CSE222
(q for quit)edge destination = CSE321
(q for quit)edge source = CSE211
(q for quit)edge destination = CSE321
(q for quit)edge source = CSE321
(q for quit)edge destination = CSE422
(q for quit)edge source = q
Adjacency List Representation of graph:
CSE102 --> ['CSE241']
CSE241 --> ['CSE222']
CSE222 --> ['CSE321']
CSE321 --> ['CSE422']
CSE211 --> ['CSE321']
CSE422 --> []
Topological Sort Results:

dfs based topological order: CSE211 -> CSE102 -> CSE241 -> CSE222 -> CSE321 -> CSE422
non-dfs based topological order: CSE211 -> CSE102 -> CSE241 -> CSE222 -> CSE321 -> CSE422
```

```
Test Menu:
1. Input edges
2. Input exponent and base
3. Generate Sudoku Board and solve
Enter 'q' to quit
Enter your selection: 2
Enter base: 3
Enter exponent: 4
81

Test Menu:
1. Input edges
2. Input exponent and base
3. Generate Sudoku Board and solve
Enter 'q' to quit
Enter your selection: 2
Enter base: 5
Enter exponent: 4
625
```

this is for input = 2



this is for input = 3

# Explanation & Analysis of The Algorithms and Solutions

## Q1-a) DFS based Topological Sort Algorithm:

### Explanation:

Initially I needed a graph implementation which must be dynamic. It is simple in python with using a hash map(dictionary) that holds vertex as key, destination list as value. It is possible to add vertex and add edges to that abstract data type. This graph could be implemented as adjacency matrix, but in this way, it is an adjacency list implementation.

After creating DynamicGraph class, I decided to make graph algorithms member functions of that class. Each algorithm will use adjList variable on that class. Firstly, dfs based algorithm is implemented. I thought if I made a dfs, and during each visitation, if vertex is added to the list, it will be topologically sorted simply. But the problem is, there could be nodes which have no incoming edges. So simple solution, requires updating. Insert a main loop which

iterates through all vertices, then if current one is not visited, make depth first search to that node. In this way it is guaranteed that, the vertices which do not have incoming edges will definitely be visited.

For the implementation details of the algorithm, visited vertices are held in a set. In this way when a node is checked for visited or not, it is done in constant time because of the hashing. I used deque for holding ordering, because inserting the elements to the node must be done from head. Deque structure does this in constant time. I also made the recursive part of the algorithm sub procedure of main order function. Additionally, graph implementation does not check if graph is DAG or not, it is assumed that graph is DAG. I did not do this check because of the aim of the assignment is more related with the algorithms.

### Worst-Case Analysis

The algorithm visits every vertex, and every edge of each vertex. If there are V vertices of the graph, and E edges of that graph. It means that, it takes linear time. Which is total number of vertices and edges.

### O(V+E)

## Q1-b) Non DFS based Topological Sort Algorithm:

### Explanation:

In this part I used same graph implementation and decided to make this algorithm to member function similar to the part 1. In this algorithm, if we are not using depth first search, then first thing to do is finding the nodes which have no incoming edges. If these edges will be visited first, then topological sort condition will not be violated. But I had a problem after that part. Problem is how to proceed, after inserting the nodes which do not have incoming edges to the head of the ordering list. These vertices can be hold in a stack. And each ordering, the element which is on the top of the stack could be appended to the ordering, but I cannot figure out how to order the rest of the vertices. I stuck on that problem, then I find a theorem which solves exactly that. It is called “Kahn’s Algorithm”. It simply removes the visited nodes from graph. And decreases edges, for every neighbor to that vertex. Eventually, some of these vertices, have zero incoming edge, and it means that they can be added to the stack, then ordering.

For the implementation details, this algorithm is based on loop as explained above. Visited vertices set is also needed here for holding every vertex in hashed set. Stack is used for consuming the vertices which have no incoming edges. Incoming edges are held in a hash map(dictionary). Vertex as key, and number of incoming edges as a value. In this way its value accessed in constant time. I hold ordering as a string in this case, and it is printed with arrows. Each visitation, it is concatenated with the vertex.

### Worst-Case Analysis:

It is similar to the previous algorithm, every vertex, and edge is visited. To find number of incoming edges for every vertex, V+E iteration is done. To add edges to the stack which have

no incoming edges, V iteration is done. Main loop, goes for every edge, and vertex also. So, there is V+E+V+V+E iteration. Which means algorithm is linear time algorithm.

### O(V+E)

### Test question 1:

```
g = DynamicGraph()
g.addEdge("CSE102", "CSE241")
g.addEdge("CSE241", "CSE222")
g.addEdge("CSE222", "CSE321")
g.addEdge("CSE321", "CSE422")
g.addEdge("CSE211", "CSE321")

g.printAdjList()
g.DFSTopologicalSort()
g.nonDFSTopSort()
```

```
CSE102 --> ['CSE241']
CSE241 --> ['CSE222']
CSE222 --> ['CSE321']
CSE321 --> ['CSE422']
CSE422 --> []
CSE211 --> ['CSE321']
dfs based topological order: CSE211 -> CSE102 -> CSE241 -> CSE222 -> CSE321 -> CSE422
non-dfs based topological order: CSE211 -> CSE102 -> CSE241 -> CSE222 -> CSE321 -> CSE422
```

## Q2) Exponential Calculation Algorithm

### Explanation:

First thought to calculate exponantiation is, multiplying number with n times, which leads to linear time algorithm. Making this logarithmic requires to eliminate half of the iteration for each number. I thought this as a recursive algorithm. Base case must be definitely the case that exponent is 0. In this case returned result will be 1. Every recursive call is done according to if exponent is even or not. If exponent is even it can be said that $a^n = a^{n/2} * a^{n/2}$ . On the other hand, if exponent is odd, $a^n = a * a^2 * a^{(n-1)/2}$. So these are the cases, with every call half of the iterations are eliminated. Because of n/2, and (n-1)/2. If this was a regular linear time loop implementation, it would be iterated n times, with this change it will eliminate the unnecessary computation.

Example case for 2^8:

2^8

(2*2)^(8/2) = (4)^4

(4*4)^(4/2) = (16)^2

(16*16)^(2/2) = (256)^1

256 * (256)^(1/2) = 256 * 256^0 = 256 * 1 = 256

### Analysis:

Algorithm eliminates half of the computations in every call with dividing exponent to 2. If this was a brute force algorithm there will be no elimination, and it would lead to linear time algorithm. With this elimination it goes to O(n) from O(logn). For every call, if base case is not reached, n/2 causes to the movement to the base case. Also, if number or exponent is 0, best case occurs according to my algorithm. It will be constant in this case.

### O(logn)

### Test for question 2:

```
#test question 2

print("\nQuestion 2 test-------\n")
print("2^5 = ", logExp(2,5))
print("3^6 = ", logExp(3,6))
print("-5^3 = ", logExp(-5,3))
print("10^4 = ", logExp(10,4))
print("2^0 = ", logExp(2,0))
```

```
Question 2 test-------

2^5 =  32
3^6 =  729
-5^3 =  -125
10^4 =  10000
2^0 =  1
```

## Q3) 9x9 Brute Force Sudoku Solver

### Explanation:

I thought brute force solution to this like; if a valid move is done, then proceed. If there is a situation occurred that board becomes unsolvable, then my previous moves must be mistaken. After that, trace back for just one move, then proceed. Do this till board is solved. It is a brute force approach, because it exhaustively checks all possible moves to reach the end. It just takes one step back for each failure. There aren't any algorithm in this approach to make smart moves etc.. It directly tries all possible moves.

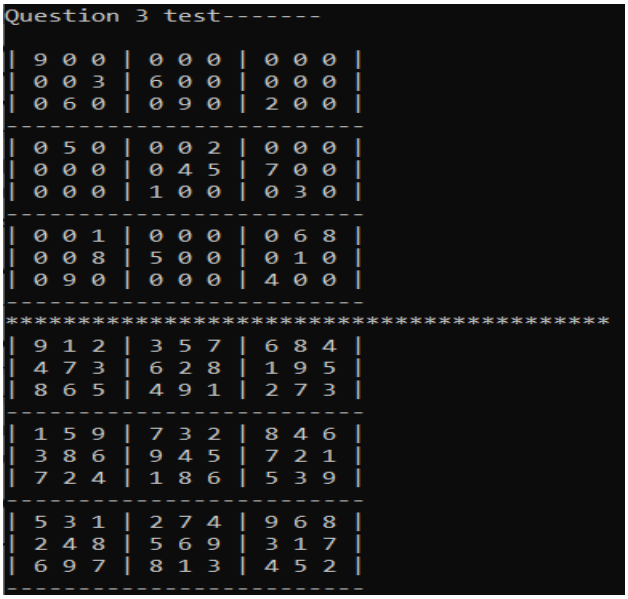
For the implementation detail, I used an Object Oriented Approach. Created a Sudoku9x9 class, and all methods, board encapsulated on Sudoku9x9 class. There is a solve() function which uses this backtracing algorithm. Initially it finds empty location on the board to put value, if there is no empty location, this means that board is solved. This is the base case of that recursive algorithm. Then tries to put the numbers between 1-9, if position is valid to put that number, puts the number. Then if board is solved, proceeds. If not, takes back that move. If board is not solved means, proceeding is done and, this move leads to a dead end, so take it back.

## Analysis:

Time complexity of this algorithm must be huge at the first glance. Because there are 81 cells, and algorithm exhaustively tries every possible combination till it is solved. Number of the empty cells are important, because each empty cell means there are 9 different value for that cell. If there are E empty cells, this means there are 9^E possibilites. Since the algorithm makes exhaustive search, and checks all posibility till it is solved. It can be said that this is the upper bound for the time complexity.

## $O(9^E)$

## Test for question 3:




# Q4) Sorting array = {6,8,9,8,3,3,12}

# Insertion Sort:

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

{6,8,9,8,3,3,12} – 8 to {6}

{6,8,9,8,3,3,12} – 3 to {6, 8}

{3,6,8,9,8,3,12} – 9 to {3, 6, 8}

{3,6,8,9,8,3,12} – 8 to {3,6,8,9}

{3,6,8,8,9,3,12} – 3 to {3,6,8,8,9}

{3,3,6,8,8,9,12} – 12 to {3,3,6,8,8,9}

{3,3,6,8,8,9,12} – sorted

Insertion sort is stable algorithm, as it can seen the same values 3s and 8s ordering remains same. It is indicated with colors. Blue ones always, remains right side on the red ones.

## Quick Sort: (middle element is chosen as pivot)

{6,8,9,8,3,3,12} – 8 is pivot

{6,8,9,3,3,12,8} – move pivot to the end, and start rearranging

{6,3,3,8,9,8,12} – after rearrange

{6,3,3} – choose left sub array pivot, which is 3

{3,3,6} - after rearranging

{3,6} – sort right sub array

{3,3,6} - after rearranging

{9,8,12} – sort right sub array, pivot is 8

{9,12,8} – move pivot to the end

{8,12,9} – choose next pivot which is 12

{12, 9} – rearrange

{9,12} – after rearranging

{3,3,6,8,8,9,12} – sorted

This is not a stable sort algorithm, as indicated 3s and 8s order is changed. At first blue ones right side of the red ones, after sorting this becomes opposite. Therefore this is not a stable sort algorithm.

## Bubble Sort:

{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}

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

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

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

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

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

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

8’s and 3’s ordering remained same for each other. Order of the elements which have same values does not changed as indicated with colors. Blue ones always remained right side of the red ones. Therefore this is a stable algorithm.

# Q5)

a) Brute Force and Exhaustive Search
   - Brute force is more general, it is an approach that tries all possibilites. They look like similar because both of the algorithms are guaranteed to find solution no matter how much time it takes. Both algorithm considers all possibilities, but exhaustive search is more specialized. Exhaustive search tries to consider all possibilites with constraints, and systematic manner. Brute force is much more straightforward and obvious. It is highly possible for some problems that brute force performs too many unnecessary operations. So, brute force is general according to the exhaustive search. Exhaustive search is a type of brute force, but as it is said it is more specialized, as a result more efficient algorithm. It can be designed for more spesific problems.

b) Ceasar’s Cipher and AES
   - They are both encryption algorithms, AES is the more secure one. On the other hand, Ceasar’s Cipher is easy to implement.
   - Ceasar’s Cipher is based on shifting the letters by a fixed number. AES is based on more complex mathematical operations. It uses secret key to encrypt and decrypt. Ceasar’s Cipher is less secure in this case, because if somebody know shifting value, it is easily encrypted.
   - For Ceasar’s Cipher if attacker tries all shift values, eventually it is solved by brute force approach. On the other hand, AES can be also vulnerable because, in this case attacker can try brute force attack to the secret key. But it is obvious that finding the secret key which is generated by unknown mathematical operations are much more harder than finding the shifting value.

c) Naive Solution of Primality Testing
   - In that approach number of divisiors determine the number of operation. If given number is big, number of division operation can change unexpectedly. For polynomial algorithms, number of operations must be proportional to the size of input. But in this case, it is not proportional. It is not polynomial because of the bit representation also. Problem size becomes the number of the bits, and it leads to exponential increasing. Number N can be represented with logN bits. In this case logN becomes problem size. If logN = k, then $2^k = N$. O(N) - > O ($2^k$). As a result time complexity grows exponentially.