GTU Department of Computer Engineering
CSE 321 – Fall 2022
Homework #3

Hasan Mutlu
1801042673

About Driver Function: The user can modify the default values in the driver function in order to give different inputs.

1) Topological Sort

Firstly, I need to explain the graph structure I built and tested my algorithm. It consists of two arrays. First array is a 1-D array containing vertices whereas the second array is a 2-D array containing edges from vertices[i]. Edges from a vertex to another vertex are are saved as follows.

a) DFS Solution:

It consists of two methods. The main method is a wrapper method. It takes a graph as parameter. It creates an array to keep track of visited vertices. Also, it creates an output array to put the vertices in topological order. Then, it checks every vertex and calls the recursive method if the vertex hasn't been visited yet.

The recursive function takes the graph, index of the current vertex, list of visited vertices and the output array as parameters. It firstly marks the current vertex as visited. Then, it checks the edges from current vertex and visits all of the destination vertices recursively.

If a vertex has no edges starting from it or all of its edge destinations are visited, it is appended to the output array. By this way, all of the vertices are appended to the output array starting from the deepest vertex to the top vertices. The array is printed backwards to display the topologically sorted graph.

Note: It doesn't check if the graph is DAG or not.

Worst Case Time Complexity:

The wrapper method iterates all the members of the vertices list. The recursive method is called for every edge on the graph. So, in any case, it takes Teta(N + V) time where N is the total number of vertices and E is the total number of edges.

b.) Non-DFS Solution

It is one method. It takes a graph as parameter. Firstly, it creates a *directed* array and fills this array according to how many edges are directed to the vertex at the index i. Then, it creates a queue and adds vertices with no edges directed to them. These vertices will be the starting vertices for sure.

After that, a BFS is performed until the queue gets empty. Next element in the queue is popped and added to the output array. All vertices that have an edge from the current vertex (the one that is popped from queue) get their directed value decremented by one as one of the vertices that direct an edge to them has been visited. If any of these vertices' directed value gets to 0, it is added to the queue. This procedure is repeated until the queue gets empty and all vertices are covered.

After that, the output array is printed and result is displayed.

Worst Case Time Complexity:

Similarly to DFS solution, every vertex and every edge is covered once so it takes Teta(N + V) time where N is the total number of vertices and E is the total number of edges.

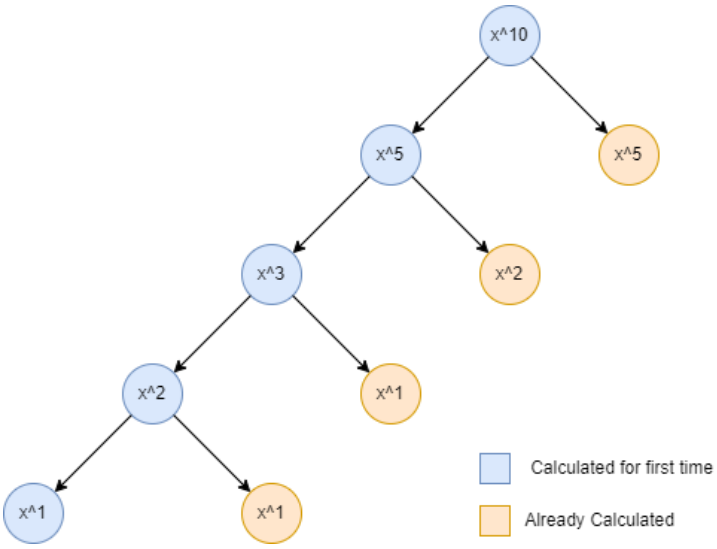
2.) Power Algorithm

Power algorithm uses dynamic programming approach. It consists of two methods. The main method is a wrapper method. It takes the number and its desired n power as parameters. Then, it creates a memory array of size ceiling(n/2) to save the previously calculated results. Then, it calls the recursive power functions for the number's ceiling(n/2), floor(n/2) powers and returns the result of their multiplication.

The recursive method takes the number, its required n power and the memory array. It first checks if the nth power of the number has already been calculated. In order to do it, it checks if the result is present in the memory array's nth index. If it is, it is directly returned as there is no need to calculate it again. If not, firstly, special cases are checked (1^n or x^0). If there is no special case, it calls itself again with the number's ceiling(n/2), floor(n/2) powers and assigns the multiplication of them into the memory array's nth index and returns the result.

Worst Case Time Complexity:

In any case, the value n is halved in every recursive call so that it takes Teta(logn) time. However, it has space complexity overhead. It uses Teta(n) amount of additional memory space.



For x = 2, memo = [0, 2, 4, 8, 0, 32]

The recursive function has been called only for the non-zero values on the array.

3.) Sudoku Solver

It consists of two methods. The main method sudokuSolver is a back-tracing recursive method. It takes the grid as parameter. It first finds the indices of first blank space in the grid. If there isn't any, it returns True. It iterates through numbers from 1 to 9 and checks if a number can be placed on that cell by calling the checkNum method.

The second method checkNum is just an utility method. It takes the grid, row, column and the number as parameters and checks if a number can be placed on that cell according to sudoku rules. If it can placed, it returns True. Otherwise, it returns False.

After placing a candidate number in the cell, it calls for the sudokuSolver method recursively to fill the next blank cell. If none of all 9 numbers can't be placed on that cell, it returns False. When it returns False, it means that one or more of the predecessor methods have placed (a) wrong candidate(s) in their corresponding cell. When the called method returns False, its predecessor method reverts its changes and tries to place next appropriate candidate number in the cell. Eventually, going forwards and backwards, all called methods would place the correct number in their cell and recursively return True all the way to the first called method.

Worst Case Time Complexity:

Assume the grid size is N and the number of blank cells is M. In this algorithm, worst case would happen when all N candidates for a cell are needed to be checked in all M blank spaces. That would make a time complexity of Teta(N^M).

4.) Sortings

a.) Insertion sort

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

- Key = 8 {6, **8**, 9, 8, 3, 3, 12}
- Compare 8 with 6 (stay) {6, **8**, 9, 8, 3, 3, 12}
- Key = 9 {6, 8, **9**, 8, 3, 3, 12}
- Compare 9 with 8 (stay) {6, 8, **9**, 8, 3, 3, 12}
- Key = 8 (right) {6, 8, 9, **8**, 3, 3, 12}
- Compare 8 with 9 {6, 8, 9, **8**, 3, 3, 12}
- Swap {6, 8, **8**, 9, 3, 3, 12}
- Compare 8 with 8 (stay) {6, 8, **8**, 9, 3, 3, 12}***
- Key = 3 (left) {6, 8, 8, 9, **3**, 3, 12}
- Compare 3 with 9 {6, 8, 8, 9, **3**, 3, 12}
- Swap {6, 8, 8, **3**, 9, 3, 12}
- Compare 3 with 8 {6, 8, 8, **3**, 9, 3, 12}
- Swap {6, 8, **3**, 8, 9, 3, 12}
- Compare 3 with 8 {6, 8, **3**, 8, 9, 3, 12}
- Swap {6, **3**, 8, 8, 9, 3, 12}
- Compare 3 with 6 {6, **3**, 8, 8, 9, 3, 12}
- Swap {**3**, 6, 8, 8, 9, 3, 12}
- Key = 3 {3, 6, 8, 8, 9, **3**, 12}
- Compare 3 with 9 {3, 6, 8, 8, 9, **3**, 12}
- Swap {3, 6, 8, 8, **3**, 9, 12}
- Compare 3 with 8 {3, 6, 8, 8, **3**, 9, 12}
- Swap {3, 6, 8, **3**, 8, 9, 12}
- Compare 3 with 8 {3, 6, 8, **3**, 8, 9, 12}
- Swap {3, 6, **3**, 8, 8, 9, 12}
- Compare 3 with 6 {3, 6, **3**, 8, 8, 9, 12}
- Swap {3, **3**, 6, 8, 8, 9, 12}
- Compare 3 with 3 (stay) {3, **3**, 6, 8, 8, 9, 12}***
- Key = 12 {3, 3, 6, 8, 8, 9, **12**}
- Compare 12 with 9 (stay) {3, 3, 6, 8, 8, 9, **12**}
- Finish {3, 3, 6, 8, 8, 9, 12}

This is a stable algorithm. The values are only swapped when one is greater than other. Therefore, the elements with the same value stayed in the same relative side of each other. The one that had lower index still has the lower index at the end.

b.) Bubble Sort

- Compare 6 with 8 (stay) {**6**, **8**, 9, 8, 3, 3, 12} exchanges = false
- Compare 8 with 9 (stay) {6, **8**, **9**, 8, 3, 3, 12}
- Compare 9 with 8 {6, 8, **9**, **8**, 3, 3, 12}
- Swap {6, 8, **8**, **9**, 3, 3, 12} exchanges = true
- Compare 9 with 3 {6, 8, 8, **9**, **3**, 3, 12}
- Swap {6, 8, 8, **3**, **9**, 3, 12}
- Compare 9 with 3 {6, 8, 8, 3, **9**, **3**, 12}
- Swap {6, 8, 8, 3, **3**, **9**, 12}
- Compare 9 with 12 (stay) {6, 8, 8, 3, 3, **9**, **12**}
- Exchanges = true so continue
- Compare 6 with 8 (stay) {**6**, **8**, 8, 3, 3, 9, 12} exchanges = false
- Compare 8 with 8 (stay) {6, **8**, **8**, 3, 3, 9, 12}***
- Compare 8 with 3 {6, 8, **8**, **3**, 3, 9, 12}
- Swap {6, 8, **3**, **8**, 3, 9, 12} exchanges = true
- Compare 8 with 3 {6, 8, 3, **8**, **3**, 9, 12}
- Swap {6, 8, 3, **3**, **8**, 9, 12}
- Compare 8 wtih 9 (stay) {6, 8, 3, 3, **8**, **9**, 12}
- Compare 9 with 12 (stay) {6, 8, 3, 3, 8, **9**, **12**}
- Exchanges = true so continue
- Compare 6 with 8 (stay) {**6**, **8**, 3, 3, 8, 9, 12} exchanges = false
- Compare 8 with 3 {6, **8**, **3**, 3, 8, 9, 12}
- Swap {6, **3**, **8**, 3, 8, 9, 12} exchanges = true
- Compare 8 with 3 {6, 3, **8**, **3**, 8, 9, 12}
- Swap {6, 3, **3**, **8**, 8, 9, 12}
- Compare 8 with 8 (stay) {6, 3, 3, **8**, **8**, 9, 12}***
- Compare 8 with 9 (stay) {6, 3, 3, 8, **8**, **9**, 12}
- Compare 9 with 12 (stay) {6, 3, 3, 8, 8, **9**, **12**}
- Exchanges = true so continue
- Compare 6 with 3 {**6**, **3**, 3, 8, 8, 9, 12} exchanges = false
- Swap {**3**, **6**, 3, 8, 8, 9, 12} exchanges = true
- Compare 6 with 3 {3, **6**, **3**, 8, 8, 9, 12}
- Swap {3, **3**, **6**, 8, 8, 9, 12}
- Compare 6 with 8 (stay) {3, 3, **6**, **8**, 8, 9, 12}
- Compare 8 with 8 (stay) {3, 3, 6, **8**, **8**, 9, 12}***
- Compare 8 with 9 (stay) {3, 3, 6, 8, **8**, **9**, 12}
- Compare 9 with 12 (stay) {3, 3, 6, 8, 8, **9**, **12**}
- Exchanges = true so continue
- Compare 3 with 3 (stay) {**3**, **3**, 6, 8, 8, 9, 12}*** exchanges = false
- Compare 3 with 6 (stay) {3, **3**, **6**, 8, 8, 9, 12}
- Compare 6 with 8 (stay) {3, 3, **6**, **8**, 8, 9, 12}
- Compare 8 with 8 (stay) {3, 3, 6, **8**, **8**, 9, 12}***
- Compare 8 with 9 (stay) {3, 3, 6, 8, **8**, **9**, 12}
- Compare 9 with 12 (stay) {3, 3, 6, 8, 8, **9**, **12**}
- Exchanges = false so finish {3, 3, 6, 8, 8, 9, 12}

This is a stable algorithm as well. Like Insertion sort, the values are only swapped when one is greater than other so order of repeating elements didn't change.

c.) Quicksort

- Pivot = 6 {**6**, 8, 9, 8, 3, 3, 12}
- Find next i and j {**6**, 8, 9, 8, 3, 3, 12} i = 1, j = 5
- Swap {**6**, 3, 9, 8, 3, 8, 12}***
- Find next i and j {**6**, 3, 9, 8, 3, 8, 12} i = 2, j = 4
- Swap {**6**, 3, 3, 8, 9, 8, 12}
- Find next i and j {**6**, 3, 3, 8, 9, 8, 12} i = 3, j = 2
- Indices collided, swap pivot with arr[j]
- {3, 3, **6**, 8, 9, 8, 12}***
- LHS, Pivot = 3 {**3**, 3, 6, 8, 9, 8, 12}
- Find next i and j {**3**, 3, 6, 8, 9, 8, 12} i = 1, j = 0
- Indices collided. Swap pivot with arr[j] (Stays the same).
- RHS, Pivot = 8 {3, 3, 6, **8**, 9, 8, 12}
- Find next i and j {3, 3, 6, **8**, 9, 8, 12} i = 4, j = 2
- Indices collided. Swap pivot with arr[j] (Stays the same).
- RHS: Pivot = 9 {3, 3, 6, 8, **9**, 8, 12}
- Find next i and j {3, 3, 6, 8, **9**, 8, 12} i = 6, j = 5
- Indices collided. Swap pivot with arr[j]
- {3, 3, 6, 8, 8, **9**, 12}
- {3, 3, 6, 8, 8, **9**, 12}
- Finish {3, 3, 6, 8, 8, 9, 12}

Quicksort is not a stable algorithm. In the first swap operation marked with ***, the element 3 on the right side crossed to the left of the element 3 on the left side. Their relative positions swapped.

In this example, since pivot's index found to be the same with one of those elements, their relative positions were swapped back. But that was by chance and doesn't happen very often.

On the right part of the array, since there wasn't any elements smaller than 8, the repeating 8's didn't change their positions as well. So, for this example, result is found to be stable but that doesn't necessarily prove that the Quicksort is stable.

5.) Questions

a.) Relation Between Brute Force and Exhaustive Search

Brute Force is the simplest, straight forward approach to solving a problem. It implies trying every different probability on the given data in order to find the solution.

Exhaustive search is a brute-force approach to combinatorial problems. This algorithm focuses on finding each solution to the given problem by satisfying all the constraints. For example, in Travelling Salesman problem, the aim is not only to find an Hamiltonian path but also to find the shortest one.

b.) Caesar's Cipher and AES

Caesar's Cipher is vulnerable against brute force attacks. There are only a limited number of (letters in the alphabet) shifts and all possibilities can easily be checked using a brute force attack.

Theoretically, AES can also be broken using brute force by trying every possible key but it would take impractically long times (thousands of years) with the current technology.

c.) Primality Testing

Because the modulo operation doesn't take constant time in the hardware. When the input size is measured as bits, it would be log(n). So the primality test is exponential in log(n)