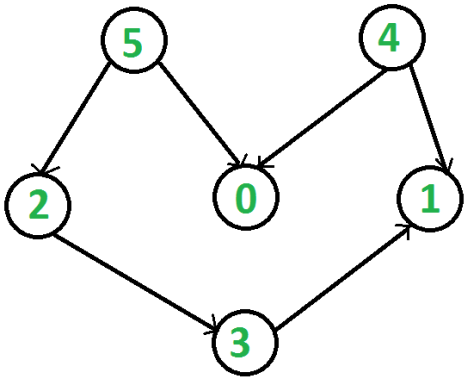
CSE 321 - Introduction to Algorithm Design

Homework 3 – Report

Ömer Faruk Bitikçioğlu

161044010

1-a) In DFS every vertex in a graph should be visited by going into the deepest level of a selected vertex. If the graph is like below:



In DFS, it should select the first vertex (in degree of 0) and go into the deepest level like $5 \rightarrow 2 \rightarrow 3 \rightarrow 1$, there is no direction to go left, so it switches the other side $\rightarrow 0$, again there is no direciton to go, no more direction for vertex 5, it selects the other vertex with the same in-degree of vertex 5 which is 0 is vertex 4. No more unvisited vertex for vertex 4. It stops.

To sum up, DFS result of this graph is : 5, 2, 3, 1, 0, 4 which is not a topological sorting. We should change the DFS algorithm for topological sorting.

DFS Based Algorithm

It should traverse the graph like DFS but it should not print the visited vertices immediately. It should hold a stack, and add vertices to stack whenever no more direction left for the vertex. For the same graph above:

$0 \rightarrow$

$1 \rightarrow$

$2 \rightarrow 3$

$3 \rightarrow 1$

$4 \rightarrow 0, 1$

$5 \rightarrow 2, 0$

Stack = []

Visited = [F, F, F, F, F, F]

We also should hold an array of visited vertices. Visit from the 0^{th} vertex to the last vertex:

Visit Vertex 0. Vertex 0 has no direction to go. Push it to stack.

Stack = [0]

Visited = [T, F, F, F, F, F]

Visit Vertex 1. Vertex 1 has also no direction to go, push it to the stack.

Stack = [0, 1]

Visited = [T, T, F, F, F, F]

Visit Vertex 2. Vertex 2 is prerequisite of Vertex 3.

Visit Vertex 3. Vertex 3 is prerequisite of Vertex 1.

Visit Vertex 1. Vertex 1 is already visited. Push 3, and 2 to the stack.

Stack = [0, 1, 3, 2]

Visited = [T, T, T, T, F, F]

Vertex 3 is already visited.

Visit Vertex 4. Vertex 4 is prerequisite of Vertex 0 and Vertex 1.

Vertex 0 and Vertex 1 are already visited. Push Vertex 4 to the stack.

Stack = [0, 1, 3, 2, 4]

Visited = [T, T, T, T, T, F]

Visit Vertex 5. Vertex 5 is prerequisite of Vertex 2 and Vertex 0.

Vertex 2 and Vertex 0 are already visited. Push Vertex 5 to the stack.

Stack = [0, 1, 3, 2, 4, 5]

Visited = [T, T, T, T, T, T]

No more vertex to visit. Print the stack in reverse order to see the topological order of the graph calculated using DFS approach:

Topological order of given graph: 5, 4, 2, 3, 1, 0

Worst Time Complexity

Worst time complexity of the given problem is O(V+E). Because we visit every each of vertex and edge once every time.

1-b) Another solution for this problem is BFS-based algorithm called Kahn's algorithm. This algorithm is based on the fact that a DAG G, has at least one vertex with in-degree 0 and one vertex with out-degree 0. So we can start with the vertices with no incoming edges since they have no prerequisite. After completing these task we can visit the vertices(jobs) has incoming edges with 1, since the prerequisite jobs are completed. After them we can visit the next in-degree level of vertices. This algorithm goes like that.

BFS Based Algorithm (Kahn's Algorithm)

Step-1: Compute the in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.

Step-2: Pick all the vertices with in-degree as 0 and add them into a queue.

Step-3: Remove a vertex from the queue (Dequeue operation) and then:

1. Increment the count of visited nodes by 1.
2. Decrease in-degree by 1 for all its adjacent nodes.
3. If the in-degree of adjacent nodes is reduced to zero, then add them to the queue.

Step 4: Repeat Step 3 until the queue is empty.

Step 5: If the count of visited nodes is not equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

Worst Time Complexity

O(V+E). The outer for loop will be executed V number of times and the inner for loop will be executed E number of times.

2) One way to calculate a^n with a worst-case time complexity of O(logn) is to use the binary exponentiation algorithm. This algorithm works by repeatedly squaring a and halving n until n is equal to 1. At each step, if n is odd, we also multiply the result by a. This allows us to calculate a^n in O(logn) time because each step reduces the value of n by half, so the total number of steps is at most logn.

3) In our algorithm to solve the Sudoku puzzle, we use a helper function named “possible” which is checking every row and column of the given vertex (x,y) and every vertex in the sub-grid of the given vertex to exhaustively search for if the given value of n is suitable for that vertex. If the row, column, or the sub-grid has that value already, it is not suitable to put the value there.

In our solve function, we solve the puzzle by using the helper function explained above. We exhaustively search all the vertices in our grid for blank vertices (value of 0) and when we find them, we try all possible values there starting from 1 to 9, if it is possible to put the value there we simply put.

We recursively call the solve function again. If we can not find a possible value for a empty space that means we did something wrong in the earlier, so we simply return. When we return we erase the value we put earlier and try other possibilities.

When we can not find a blank vertex in our grid, that means we solved the puzzle and we can print it. If there is more possibilities it will go back again (like the situation it mistakenly put a wrong value) and try other possibilities. If it finds another possible solution it will print it too.

Worst Time Complexity

Time complexity depends on the grid size. Let’s say grid is NxN matrix and we would try N possibilities for each vertexes in a worst case scenario. Worst time complexity is then O(n^(nxn)).

4) To sort the array using insertion sort, we first compare the first two elements, 6 and 8. Since 8 is larger than 6, we leave the array as it is. Next, we compare 8 to 9 and again, since 9 is larger, we leave the array unchanged. We then compare 9 to 8 and since 8 is smaller than 9, we shift the 9 to the right. Then we compare 8 with 8 and since they are equal we leave it and place the 8 there. The array is now [6, 8, 8, 9, 3, 3, 12].

Next, we compare 9 to 3 and since 3 is smaller, we shift the 9 to the right. Then we compare 3 with 8, 8, and 6, all of them are bigger than 3 and we shift them. No element left to compare and we put the 3 at the beginning. The array is now [3, 6, 8, 8, 9, 3, 12]. We then compare again 3 to sorted part(left part of the array). Shift the bigger elements to the right and put the 3 next to other 3. The array now looks as [3, 3, 6, 8, 8, 9, 12]. We then compare 12 to the sorted part, whole numbers are smaller than 12 so we leave it that way. The array is now sorted as [3, 3, 6, 8, 8, 9, 12].

The resulting array after using insertion sort is [6, 3, 3, 8, 8, 9, 12]. Insertion sort is a stable sorting algorithm because it maintains the relative order of the elements with equal values.

To sort the array using quick sort, we first choose the first element, 6, as the pivot. We then partition the array into two sub-arrays, one with elements less than or equal to 6 and the other with elements greater than 6. The resulting arrays are [3, 3] and [8, 9, 8, 12]. We then apply quick sort recursively to each of the sub-arrays. The final sorted array is [3, 3, 6, 8, 8, 9, 12].

Quick sort is not a stable sorting algorithm because it does not maintain the relative order of elements with equal values. In the above example, the two 3's were not in the same order in the original array, but after sorting with quick sort, they are next to each other.

To sort the array using bubble sort, we first compare the first two elements, 6 and 8, and since 8 is larger, we leave it as it is. Next we continue with the next items one by one comparing and swapping if the left part is bigger. The array is now [6, 8, 8, 3, 3, 9, 12]. We then compare the next elements, we swap the two elements if the left part is bigger. We repeat this process until the array is fully sorted.

The final sorted array using bubble sort is [3, 3, 6, 8, 8, 9, 12]. Bubble sort is a stable sorting algorithm because it maintains the relative order of the elements with equal values. For example, the two 3's were next to each other in the original array and they remain next to each other in the final sorted array.

5-a) Brute force and exhaustive search are closely related terms that are often used interchangeably. Brute force refers to a problem-solving approach that uses a simple and straightforward method to try all possible solutions to a problem. It is a straightforward, but often inefficient, method of solving a problem by trying every possible solution until the correct one is found.

Exhaustive search is a specific type of brute force algorithm that considers every possible combination of inputs to a problem in order to find the correct solution. It is a systematic and thorough method of searching through all possible solutions to a problem in order to find the correct one.

In summary, brute force is a general term that refers to using simple and straightforward methods to solve a problem, while exhaustive search is a specific type of brute force algorithm that considers every possible combination of inputs to a problem. Both methods are simple and straightforward, but can be computationally expensive and time-consuming for large or complex problems.

5-b) Caesar's Cipher is a simple substitution cipher that is based on the idea of replacing each letter in a message with a different letter. It is named after Julius Caesar, who is said to have used this cipher to communicate with his generals. The cipher works by shifting each letter in the message by a fixed number of positions in the alphabet. For example, if the shift value is 3, then the letter "A" would be replaced with "D", "B" would be replaced with "E", and so on.

Caesar's Cipher is vulnerable to brute force attacks because it only has a small number of possible shift values (26 for the English alphabet). This means that an attacker can simply try all possible shift values until the correct one is found, which can be done relatively quickly even for long messages.

AES (Advanced Encryption Standard) is a modern symmetric key encryption algorithm that is widely used to secure sensitive information. It is a block cipher, which means that it encrypts data in

fixed-size blocks, rather than encrypting one bit at a time like some other ciphers. AES uses a key to encrypt and decrypt the data, and the security of the encryption depends on the strength of the key.

AES is not vulnerable to brute force attacks because it uses keys that are much longer and more complex than the keys used in Caesar's Cipher. The length of the key determines the number of possible keys that the attacker must try in order to break the encryption, and for AES, this number is so large that it is computationally infeasible to try all possible keys. This makes AES much more secure than Caesar's Cipher, and it is widely used for encrypting sensitive data.

5-c) The naive solution to primality testing, which involves checking if x $\in$ {2, 3, ..., n - 1} divides n, grows exponentially because the number of values of x that must be checked increases exponentially as n gets larger. This is because the number of values in the set {2, 3, ..., n - 1} grows exponentially as n increases.

For example, if n = 10, then the set {2, 3, ..., n − 1} contains 9 values, and the algorithm must check each of these values to see if they divide n. If n = 100, then the set {2, 3, ..., n − 1} contains 99 values, and the algorithm must check each of these values to see if they divide n. As n gets larger and larger, the number of values in the set {2, 3, ..., n − 1} grows exponentially, and the algorithm must check an exponentially increasing number of values to determine whether n is prime or not.

This makes the naive solution to primality testing impractical for large values of n because the computation time grows exponentially with n. More efficient algorithms, such as the Sieve of Eratosthenes, have been developed to solve the primality testing problem in a more efficient manner.