Ryan Johnson

BI-GY 7453: Prof. Manpreet S Katari

May 3, 2022

**ASSIGNMENT 5 (Final Assignment) : Due Date : 10-May-2022**

**Read it before attempting :**

1. Provide clear,lucid explanation for each question
2. Wherever possible used a small code snippet (with output) to better explain your answer
3. All question are form class lectures and slides, So go through them before attempting any questions

**Question 1.** What does it mean when an algorithm is P or NP complete?

When an algorithm is P or NP complete, that refers to the ability of said algorithm to solve a problem in a certain time frame. This means an algorithm that can deterministically find a solution in polynomial time, would be considered P. The algorithm is considered to be NP if there is no known time in which it would deterministically be able to solve the problem if the answer is not already known. NP gets a bit more complicated, where you need to look at if the NP problem can be verified in polynomial time. If you can find a verifier that runs in polynomial time, then it can be considered NP. There is a popular term in computer science called P versus NP. This is where scientists are looking for an answer to prove P=NP. Currently the known idea is P is not equal to NP, because there is no algorithm where an NP problem that can be verified in P time and can also be solved in P time. Some problems can also be NP-hard or NP-complete. NP-complete being where many of the problems that have no algorithm that can find an easy solution to it reside.

**Question 2.** What is the difference between Big O, Big Omega, and Big Theta functions?

Big O is mainly about finding an asymptotic upper bound, while omega is referring to finding the lower bound, and finally theta is looking at the exact bounds (lower and upper) of an algorithm. Given this nature, big theta is contained within big O and big omega and thus, a function considered to be big theta can also be considered big O. This case is not reverse, where a function can be big O but not big theta. When looking at big O, since it is considered to be upper bound, an algorithm with this notation would require the most time to compete its calculations leaving us with the worst-case performance. When looking at big omega, being lower bound, this means the algorithm would require the least time thus meaning it is the most efficient way to solve a problem. When looking at big theta, being the exact bound, this means that the algorithm would take the best of the worst time to solve a problem.

**Question 3.** Explain the difference between an adjacency list and an adjacency matrix. How do they store networks and in what scenarios is one preferred over the other?

An adjacency list is the way of representing points on a graph in a compacted format and are better for sparse graphs. Adjacency lists are helpful to use when trying to manipulate the data of nodes that are connected/related to each other, to make sense of the graph and to represent weighted graphs. Adjacency lists are exactly like dictionaries found in python. They have a key, which can be any node found in a graph, and multiple values, which would be all the other nodes that are directly connected to the key. Once graphs become very large however, it would be best to use an adjacency matrix to represent the nodes associated with the graph. Adjacency lists become a hassle to keep up with when the value connected to a single key become too many. An adjacency matrix is just a matrix that maps a list of all the nodes to both the columns and rows. The column/row pairs would be considered an edge, which means it is where a node in the row has a connection to a node found in the column. If a column/row pair does not have a connection that would be represented as a 0 in the space. If there is a connection between the column/row pair, then a 1 would be noted in the space.

**Question 4.** Give a summary of BFS and DFS. Make sure to explain the difference between them. What is the time complexity of both? Explain why this is the case.

BFS is also known as Breadth First Search, and DFS known as Depth First Search, these are algorithms used to search a graph. Algorithms like these two allow us to analyze and reorganize many different graphs to make sense of data. BFS is an algorithm which prioritizes exploring a neighborhood and works on both directed and undirected graphs. The BFS algorithm aims to find a path form a source vertex ‘s’ to a final vertex ‘v’ that is reachable then, a breadth first tree is produced in this process. When a vertex ‘v’ is reachable from ‘s’ in the breadth first tree that would be considered the shortest path in the graph between the two nodes. When implementing the BFS algorithm in python, we understand that each vertex is added or enqueued only once so the time for that would be O(|V|). Next the adjacency list of each vertex is scanned once and we know that a key value pair in the list is representative of an edge in a graph, so the time scanning the lengths of the lists would be O(|E|). When we combine these two parts, we see the BFS run time is O(|V|+|E|) which is linear time. DFS is one which prioritizes exploring a direction. So, this means that the DFS algorithm would continue searching on a path as long as there are still unexplored edges leaving a given vertex. After all the edges have been accounted for the search will backtrack from the final vertex that was found. This would continue for other vertices until there are none remaining. Each vertex in this process has two time-stamps, the first being when it is first discovered, and the second being when the search finishes examining that vertex’s adjacency list on the traceback. There are two loops in the DFS algorithm and they take big theta(|V|) time, this is because we are going through a DFS tree visiting the vertices twice as mentioned earlier. The next part of the algorithm is DFS-visit which is called once only when a vertex is colored white, the loop is done as many times as a key value pair exists in an adjacency list which would be big theta(|E|). So the running time is O(|V| + |E|) which is linear.

**Question 5.** In Lecture 7 we discussed different algorithms for string matching. Describe one and give an example of how it can used in genomics research today?

In lecture 7 we discussed an algorithm called naïve string matcher. This algorithm takes the length of the text and the length of the pattern strings and assigns them a variable. Next it will loop through a range from the variable that represents the length of the text we will call ‘ n’ minus the length of the pattern we will call ‘m’. We do this to account for the fact that the pattern, as a whole, will continue to shift one place to the right when looking for a match with the text, thus the pattern can’t shift any more once it reaches the end of the text. So ‘s’ would be the number of total times the pattern can shift through the text if no matches are found. Then within the loop we will have a conditional statement saying if the pattern is equal to the text at the position of the shift, the naïve string-matching algorithm will print that it has found a match at position ‘s’. One way this algorithm is useful in genomics research today is that we can look through a gene which would be our text, and determine if a marker of our choice, the pattern, is present in said gene. The marker we are looking for could be a specific sequence where we want to cut our gene to do further analysis on the separate pieces. Then once we find that location we would know where the CRISPR technology can perform the cut, or if the location that matches the enzyme is even present in the text in the first place. This is just a modern example where this algorithm could be of use, instead of trial and error in a wet lab we could do the analysis using code and perfect the procedure before trying it in a real setting.

**Question 6.** In Lectures 10 and 11 we discussed different algorithms for sorting. Describe one and give an example of how it can used in genomics research today?

One of the sorting algorithms discussed in lecture 10 that I want to talk about is merge sort. The merge sort algorithm is best described as using a divide and conquer method on an array consisting of ‘n’ elements. The divide portion is to divide said array into two smaller sequences which will have n/2 elements respectively. Next would be to conquer the two smaller sequences recursively using merge sort. As one would guess once we called the merge sort algorithm recursively this would produce more subsequences of each previous subsequence, until we get the smallest possible subsequences. Next from the smallest subsequences, the algorithm combines the two smaller sequences in a sorted way until it gets back to the larger array to give the final sorted output. Many of the sorting algorithms would be useful in the same way for genomic analysis. One of these uses would be to sort a large data set consisting of genes, their locations on the genome and how many have known corresponding exons located in them. We can create a file where we have a collection of chromosomes and all the known start positions associated with each. In sorting this file, we would be left with an ordered set of data which we can then pass through another large file that consists of gene annotations. Once the data is sorted it would be more efficient to write code to complete that analysis. This is because once looping through chromosomes and their start positions, we can break the code if said start positions no longer fall in the region consisting of exon ranges associated with the chromosome pair. In shorter terms, sorting genomic data would make analysis of the large data sets much faster, without sacrificing accuracy.

1