CS 330 HW #5

Chandini Toleti U29391556
Collaborators: Irene Deng

1.

(1.1A) ALGORITHM

```
Inputs:
M←2d array of words

def create graph(M):

Initialize:
G← empty nested hash table

for word in M do:
        for char in word:
                if char not in G do:
                        G[char] = []

Do:
For i=0 to len(M)-2 do:
        Word1←M[i]
        Word2 ←M[i+1]

        Range ←min(len(word1), len(word2))

        For j=0 to range-1 do:
                If word1[j] != word2[j]:
                        G[word1[j]].append(word2[j]
                        Break
Return G
```

(1.1B) PROOF OF CORRECTNESS

ASSERTION 1: Every distinct character that appear M[][] is a key in the graph G. Consider the direct proof: the algorithm loops through every word in M[] and every char for every word M[][]. If the letter is not already a key in G, the algorithm adds it.

ASSERTION 2: The algorithm assumes an alphabetical order and adds letters correctly based on this assumption. By observing words in alphabetical order, we know that we

compare the first letters in each words, then the second letters, then third, etc. if the first letters in both words aren't equal. For example, assuming 'bat' and 'cat' are in alphabetical order, we only look at the first letter and know b<c since they aren't equal. On the other hand, for 'bar' and 'bat', we compare b=b, a=a, until r!=t. At this point we know r<t, assuming 'bar' and 'bat' are in alphabetical order. Similarly, the algorithm first checks the first to the second-to-last word and compares it to the next word. For each pair of words, the algorithm checks every matching letter until an inequality is found, which in that case it would move on to the next word. Assuming an alphabetical order, this would correctly add any dependencies to the graph G.

Consider the direct proof: The algorithm stores all distinct letters as keys and only adds values if letters show evidence of being ordered, given by assertion 1. Additioanlly, the algorithm has no infinte loops as it terminates after viewing every pair of words in M. All-in-all, the algorithm correctly adds letters to a directed graph that can later be checked for topological ordering.

(1.1C) TIME/SPACE COMPLEXITY

TIME ANALYSIS To analyze time complexity, let's look at each step of the algorithm.

(1) To initialize the nested hash table, the algorithm loops through all W words and all letters (max 35) for all words. Therefore, the worst-case time complexity of this step takes O(35W) complexity.
(2) The next set of nested loops first loops through all W words. For each word, the algorithm performs one check per character, which in the worst case is 35. In total these nested loops have a time complexity of O(35W) as well.
(3) All other elementary operations run in constant time

All-in-all, the total runtime complexity of this algorithm is O(35W + 35W) = O(70W)= **O(W)** time complexity

SPACE COMPLEXITY To analyze space complexity, let's analyze the inputs, outputs, and other temporary structures.

(1)The input is a 2-D Array M, which stores W words of 35 characters each. Therefore, its space complexity is O($35W$)

(2)The output of this algorithm is a nested hash table G, that stores an adjacency list. From lecture, we know adjacency lists have space complexity O(V + E), where V is the number of vertices and E is the number of edges. In this instance, in the worst case, the number of vertices would be L, the number of distinct letters in the alphabet since they will not appear as keys twice. Finally, the number of worst-case edges is L^2 if every letter shows evidence of being ordered with every other letter. However, this is highly unlikely. In an extreme upper bound, the space complexity is given as O(L + L^2) = O(L^2). However, a much more accurate upper bound is O(L).

(3)All other elementary structures take O(1) space, so we disregard

All-in-all, our space complexity can be given as O($35W + L^2$) = O($W + L^2$)

(1.2A) ALGORITHM

```
Inputs:
G← nested hash table adjacency list that represents ordered letters

def find_alphabet (G):

Initialize:
visited ←empty hash set, these nodes have
previous ←empty hash set, stores nodes that could create cycle
alphabet ←empty list, will store valid alphabet later

        #is_valid is a modified, recursive DFS function
        def is_valid(L):
                If L in previous do:
                        return false
                If L in visited do:
                        return true

                visited.add(L)
                previous.add(L)

                for letter in graph.get(L,[]) do:
                        If is_valid(letter) == false do:
                                return false

                previous.remove(L)
                alphabet.append(L)
                return true

for letter in G.keys() do:
        If letter not in visited do:
                If is_valid(letter) == false do:
                        return “not a valid alphabet”

return topological_order[::-1]
```

(1.2B) PROOF OF CORRECTNESS

ASSERTION 1:If a graph contains a cycle, by definition it is not a topological order and cannot be a valid alphabet. To be correct, the algorithm must return "not a valid alphabet," if the graph contains a cycle.

ASSERTION 2: If a graph has a topological order, there exists at least one valid alphabet. To be correct, the algorithm must return a valid alphabet when there are no cycles present in the graph.

OBSERVATION 1: is_valid is a modified version of DFS from lecture. We observe this method explores all letters and their potential orders to build a topological order.
OBSERVATION 2: the algorithm modifies DFS by adding previous, a hash set that stores previously explored nodes in the current recursion. This means the letters in this set are ancestors of the current letter in question.
OBSERVATION 3: DFS appends items to the alphabet after a full recursive stack. This means items are appended in reverse order of discovery, which the main algorithm reverts to the order of discovery.

Consider the Direct Proof:

By observation 1, we know DFS considers every letter in the Graph. The algorithm compares check every letter to see if it appears in *previous*. By observation 2, we know that *previous* stores all ancestors of the current letter being compared. By definition of a cycle, if there exists a letter that is an ancestor of itself, there is a cycle. By definition of topological order, a valid topological order cannot contain a cycle. Since the algorithm returns false and does not append the letter to the alphabet when a cycle is found, the integrity of both of these facts is upheld.

On the other hand, if an entire recursive stack returns and never returns false, there exists no cycle. By the definition of topological order, we also know that this implies the letter can be added to the alphabet while upholding the topological order. Finally, by observation 3, we know that the recursive calls return in reverse order of discover and by the nature of graph G implies alphabet returns in reverse alphabetical order after DFS. Since we reverse the reversed order, we know the output alphabet to be a valid topological order.

(1.2C) SPACE AND TIME COMPLEXITY

To analyze time complexity, we look at each individual step of the algorithm:

(1) The outer loop considers each key of Graph G. Graph G has L keys for each distinct letter in the alphabet. Consequently, this loop has a time complexity O(L)
(2) Next, for every letter L, the algorithm calls a modified version of DFS from lecture. We know DFS will consider every edge of the Letter passed. Therfore, in total the

DFS implantation runs in O(L + E), where L is the number of distinct letters and E is the number of edges or defined orders between letters.

(3) Finally, re-ordering the alphabet string will run in linear, or O(L) time.
(4) All other elementary operations take O(1) or constant time.

All-in-all, the algorithm runs in O($L + E + L$) = O($2L + E$) = O($L + E$) time since we disregard constants.

To consider space complexity, we observe the inputs, outputs, and temporary structures used by the algorithm.

Inputs: The algorithm takes as input a nested hash table G that functions as an adjacency list. From lecture, we observe that these nested hash tables take O(L+E) space since the Graph has a vertice for each distinct letter L, and E represents the number of edges between them.

Outputs: The algorithm returns an array alphabet, which stores a valid alphabetical ordering when possible. This ordering will have at most L elements, one for each letter of L distinct letters. Consequently, this structure takes O(L) space.

Temporary Structures: The algorithm uses two Hash sets to store information: previous and visited. Both of these structures contain at most L elements, for every distinct letter of the alphabet. Therefore the two structures together have space complexity O(2V)
All-in-all, the algorithm takes O(L+E+L+L+L) = O(3L + E) = O(L+E) space complexity, combining like terms and disregarding constants.

2.

(2.2) Counter-example to greedy algorithm

Consider the counter-example. Say we have a set of band times such that

Band 1 plays [1,5]
Band 2 plays [5,7]
Band 3 plays [2,4]

In this example, the most intersections that occur are 2. Therefore, the algorithm may choose t=3 since two intersections occur. Then the algorithm would choose a time in the last set, like t=6. This means there are two times chosen, when instead they could have chosen time 7 to minimize the solution.

(2.3) Proof

By definition of mutual excusivity, two bands that are mutually independent are bands that cannot be playing at the same time. In other words, the intersection between their intervals is the empty set. Consider the inductive proof:

Base case(k=1). Say there is one band that plays. There exists at least 1 time t in its interval to see the band.

Inductive hypothesis: Say k=2. There are two bands that are mutually exclusive. As observed earlier, there exists no time t where both bands are playing. Therefore, in order to see both bands, we must select a time t from Band 1's set and another time t from Band 2's set. Therefore, the minimum number of selections is 2=k.

Inductive step: As seen for every k mutually exclusive events, there will never exist a time where we can attend both events. Therefore, if k mutually exclusive bands exist in the entire festival, there must be exactly k intervals selected to view each of these mutually exclusive events. Additionally, more event may be selected to view other intersecting events. However, it still holds true that for k mutually exclusive events, *at least* k time intervals must be selected to view all events.

Therefore, the assertion holds true by induction and the definition of mutual exclusivity.

(2.4) ALGORITHM

```
Inputs:
endpoints← 2-D array that stores the start and finish times of each interval in pairs, (i.e.
{{1,3},(2,3)} )

def good_times(endpoints):

        Initialize:
        visited_times←an array to store times we've already selected
        do:
        endpoints ←merge_sort(endpoints) #call mergesort such that the intervals are sorted by
earliest finish time

        for interval in endpoints do:
                end← endpoints[interval][1]

                if end not in visited_times do:
                        visited_times.append(end)

        return visited_times
```

(2.5) PROOF OF CORRECTNESS

Assertion 1: As observed in lecture, sorting intervals by earliest finish times lets us find the largest set of mutually exclusive events in any interval schedule.

Assertion 2: As proved in part (2.3), we know that if there exists k mutually exclusive band intervals, then at least k times must be selected.

Assertion 2 says a schedule with k Mutually exclusive events must have k selected time slots. Consequently, we want to find sets of mutually exclusive sets. We can do this in many ways, but we opt to use EFTF from lecture because this produces the set with the MOST* mutually exclusive events in the schedule.

By selecting the greatest degree mutually exclusive set, we know that for each of these sets, we will not see 2 sperate mutually exclusive events happening during a single selected set because otherwise that selected event will not be considered the highest degree mutually exclusive set.

Finally, we use the same EFTF, so we know that we will end up checking every possible mutually exclusive event. Next, we select finish times for intervals that have not been selected yet, and finally, this leaves us with a minimum set of times to see every band in the festival.

(2.6) TIME AND SPACE COMPLEXITY

TIME: To analyze for time complexity, we look at each step of the algorithm.

(1) Initializing visited_times is constant since it is an empty array O(1)
(2) We call mergeSort, which we know to have a worst-case time efficiency of O(nlogn)
(3) Finally, we iterate over every final time in each interval. So, for n intervals the efficiency is O(n)

All-in-all the overall run time can be described as O(nlogn + n)= **O(nlogn)** since we only consider the highest degree

SPACE: To analyze space complexity, we observe the inputs, outputs, and any temporary structures used running the algorithm

INPUTS: The 2-D Array storing intervals will take O(2n) space, since there are n arrays of length 2 being stored
OUTPUTS: visited_times will at most hold all n final times in the case all intervals are mutually exclusive. Therefore, the worst-case space complexity is O(n)
TEMPORARY STRUCTS: calling mergeSort will add O(n) space due to the nature of mergeSort. Additioanlly, end takes O(1) space.

All-in-all, the overall space efficiency for this algorithm can be denoted as O(2n+n+1) = **O(n)** since we disregard constants and scalars.