Name: Nathan Nguyen

Class: CPSC535 Monday 7:00 PM to 9:45 PM

**CPSC535 PROJECT 2**

**Problem 1: Optimization of production lines**

**Pseudo code:**

Define Production\_Optimization (Durations[], stations: int):

#for this function we will use the binary search to optimize the production line.

#we will define the lower bound and the higher bound to find the longest station

#the longest duration of the single station can be between the longest\_durations of a single station to the total summation of all durations in the Durations[] array.

Lower\_bound = maximum value of Durations[]

Higher\_bound = Summation of all durations in Durations[]

#define the result to return the longest duration

Result = max(“Inf”)

#Optimize using binary search. In between the lower bound and the higher bound, we calculate the longest\_estimate\_value to be the middle of the lower bound and higher bound.

While lower\_bound <= higher\_bound:

Middle = lower\_bound + ((higher\_bound – lower\_bound) // 2)

#we check if we can divide stations further by estimating if the can\_divide() is in the lower half or the upper half of values between the lower\_bound and the higher\_bound.

If can\_divide(middle): #if we can continue to divide then the solution is in the lower half , we reduce the higher\_bound by half

Result = middle

Higher\_bound = middle – 1

Else: #if not we increase the lower\_bound because we know the solution will be in the upper half between the lower bound and the higher bound.

Lower\_bound = middle + 1

Return result

#checking if the line can be optimized. A line can be optimized if the duration of the current station is less than the longest\_estimate\_value.

Define can\_divide(longest\_estimate\_value):

number\_Of\_station = 1 #initiate the number of stations

Duration\_of\_current\_station = 0 #initiate the duration of the current station

#traversing through the Durations[] and adding the duration to 1 station until we cannot add the duration (duration is greater than the longest\_estimate\_value)

For duration in durations[]:

Duration\_of\_current\_station = Duration\_of\_current\_station + duration

If (duration\_of\_current\_station > longest\_estimate\_value):

number\_of\_station = number\_of\_station + 1 #we increment the number\_of\_station to indicate this is the longest duration we could estimate for 1 station and continue to the next station.

Duration\_of\_current\_station = duration #when duration\_of\_current\_station cannot be added due to exceeding the estimated value, we start the next station with duration.

#after greedily separating the stations, we want to return to the function can\_divide by checking if the number\_of\_station is greater than our input stations.

return number\_of\_status + 1 <= stations

**Analyze and mathematically proof:**

For this problem, we can use Master theorem to prove the time complexity of this algorithm.

T(n) = r \* T() + c(n)

With r is the number of sub problem in the recursion

d is the size of each subproblem

c(n) cost of work done outside the recursive call

1. For the optimization problem, for every iteration in the while (lower\_bound <= higher\_bound) we calculate the middle and continue to search based on each half, therefore: The size of each subproblem will be divided in half [Lower\_bound, middle] and [middle, higher\_bound] => d = 2
2. After division into 2 equally sized sub regions to search. We called the function can\_divide() to check if we can further search the value. Since there is a for loops to traversing the Durations[] we have time for running this function is O(n) => c(n) = O(n)
3. In each recursion, we only divide into 2 sub problem => the number of sub problems are 2 in each recursive call => r = 2

Therefore: T(n) = 2 \*T() + O(n)

With r = 2 , d = 2, k = 1, we have r = dk (2 = 21), this fall into case 2 of the Master theorem:

O (nk log2n) = O (n1log2n).

Total time complexity of this algorithm is O(nlogn)

**Problem 2: Finding the longest string chain**

**Pseudo code:**

Define Longest\_chain(Strings: List[str]) -> List[str]:

Sort(Strings[] in non-increasing order of each words length)

Word\_hashMap= {map each word in strings[] list} #key = word and value: position in the Strings[] list

Result {to keep track of maximum\_chain\_length, string\_chain}

# Variable to keep track of the longest word chain found before storing

longest\_chain = []

# Iterate over each word and find the longest chain

for i in range(len(strings)):

\_ , current\_chain = self.Depth\_First\_Search(i, strings, word\_hashMap, result)

# Update the longest chain if a longer one is found

if len(current\_chain) > len(longest\_chain):

longest\_chain = current\_chain

# Return the longest word chain

return longest\_chain

#define the depth first search function to take in strings list and word\_hashMap to help with searching the string chain

Define Depth\_First\_Search(index, Strings: List[str], Word\_hashMap{}, Memo{}):

#If the result for the word is already computed, we can just return the result

If i in memo:

Return memo[i]

#if not then we initialize the maximum length and string\_chain starting with the current strings

Maximum\_chain\_length = 1

String\_chain = [strings[i]]

#Iterate over each character in the current word

For index in range(len(words[index])):

#we create a predecessor by removing the jth character

Predecessor = remove jth character in word[index]

#if predecessor in word\_hashMap then we recursively find the maximum chain length and also the string chain sequence for the predecessor

If predecessor in word\_hashMap:

Current\_chain\_length, current\_string\_chain = Depth\_First\_Search( word\_hashMap[Predecessor], Strings, word\_hashMap, memo)

#Update the maximum\_chain\_length and string\_chain if longer than the current maximum

If current\_chain\_length + 1 > maximum\_chain\_length:

Maximum\_chain\_length = current\_chain\_length + 1

String\_chain = [word[index]] + current\_string\_chain

# Store the result in the memo dictionary for memoization

memo[i] = (maximum\_chain\_length, string\_chain)

return memo[i]

**Analyze and Mathematical proof:**

1. Time complexity of sorting the string in non increasing order in python is O(N log N)

The main part of this algorithm is the DFS (Depth First Search) which is a recursive DFS with memorization. The DFS called for each string in list Strings[]. in the worst case, the DFS function iterates over all characters of a string and performs recursive calls for its valid predecessors.

Let M be the maximum length of a string. The inner loop runs up to M times for each string. Each recursive call involves creating a substring, which is an O(M) operation.

The memoization ensures that each unique string is processed only once. Therefore, the total work done by the DFS across all calls is bounded by the number of strings and the operations performed per string.

The time complexity for the DFS part is O(N \* M2). This accounts for iterating over each string once (factor of N) and performing O(M2) work per string due to substring operations inside the M-length loop.

Overall time complexity when combining the sorting step (O(NlogN)) and the DFS part (O(N \* M2)) is O (N \* M2)

1. We can prove time complexity using the properties of O.

In the algorithm we have: the sorting part 𝑓0(n) = O(N log N)

and also the DFS part 𝑓1(n) = O (N \* M2)

For any complexity functions f0 (n) and f 1 (n), we have this property:

𝑂(𝑓0(𝑛) + 𝑓1(𝑛)) = 𝑂(max(𝑓0(𝑛), 𝑓1(𝑛)))

N \*logN + N\*M2 = max (N \*logN, N\*M2) = O(N \* M2)

Therefore Time complexity of this algorithm is O( N \* M2)