COMP3121 Assignment 2 Q2

Q2

2.1

Consider an algorithm which allocates the items from array P to as many customers as possible such that the price of the item given to the customer i is at least $M[i]$.

The first stage of the algorithm will include initialising a new array, $M1$ which will store a copy of array M but sorted in ascending order.

The algorithm will then sort the M in ascending order of their value using a merge sort and store this into $M1$ as objects, holding $value$ and $index$. The object key $value$ will store the value of $M's$ elements and $index$ will store the original index of the given element in M. This action will have a time complexity of $O(n \log n)$ where n is the maximum length of array M as the time complexity of a merge sort is $O(n \log n)$. The worst case time complexity of adding to the new array is $O(n)$, however, this will be overpowered by $O(n \log n)$

The second stage of the algorithm will involve iterating over the customers in $M1$. For each customer, i, the algorithm uses a binary search over array P to find the jewellery, j, such that $M1[i].value \leq P[j]$. The first iteration of this binary search should search over the bounds $[0, m]$.

If there is a value found where $M1[i].value \leq P[j]$, then the algorithm can decide that $M1.index$ should be allocated jewellery j (j being the index representing the jewellery in P).

If there is not a value found for the first customer, the algorithm will go to the next customer.

Otherwise, the algorithm will iterate to the next customer i in $M1$. For the next iterations, the binary search should occur from the new bounds $[j+1, m]$. This is done because the value of j continues to update based on each piece of jewellery if it is allocated to a customer. The bound becoming smaller will benefit the algorithm and is justified because both array $M1$ and P are in ascending order. Each time the algorithm iterates to the next customer in $M1$, it can be ensured that their $value$ is either equal to or greater than the current value of $P[j]$. Each new bound begins at $j+1$ as it will account for the fact if $P[j]$ has already been allocated to an existing customer.

This loop will end once all the customers have been considered.

The time complexity of running this binary search will have a worst-case time complexity of $O(\log m)$ where m is the maximum length of array P. Additionally, this binary search will have to be run n times as that is the length of array $M1$ which gives this stage of the algorithm an overall time complexity of $O(n \log m)$.

To calculate the total time complexity of the overall algorithm, the time complexity of the first stage and the second stage are summed together, resulting in $O(n \log n) + O(n \log m) = O(n \log n + n \log m)$.

Additionally, since $n \leq m$, this time complexity can be furthered to become $O(n \log n + m \log m)$.

The time complexity can be simplified to prioritise the dominant terms, resulting in a final time complexity of $O(n \log n + m)$ as $\log m$ has a slower growth than $\log n$ in the long run and therefore can be dropped.

Thus, the final overall time complexity of this algorithm which allocates the items from array P to as many customers as possible is $O(n \log n + m)$.

2.2
Given an array $P[1..m]$ of jewellery prices, sorted in ascending order, and array $M[1..n]$ and $B[1..n]$ of customers' minimum prices and budgets, consider the an algorithm which allocates items to as many customers as possible, such that the item given to customer i is at least $M[i]$ and doesn't exceed $B[i]$, if they are given an item at all. This algorithm will also run in $O(nm)$ time.

This algorithm will begin similarly to the beginning of question 2.1 by first sorting the array M.

Firstly, initialise array $M1$ to hold the ascending sorted M array.

The algorithm will use a merge sort to sort M into a M. For this array, the elements are stored as objects with the elements $value$, $index$, and. As the name implies, $value$ will hold the value of the element from the original array and $index$ will store the correlating index of the element from the original array to keep track of the unique customer. Additionally, the $M1$ object will have an additional key called $served$ which is a boolean type set to false by default to show whether the customer already has been served an item. This will result in an overall time complexity of $O(n \log n)$.

Next, the algorithm will run a for loop. The main loop will be such that for each piece of jewellery, j, the algorithm will run a nested loop over customers i from 1 to n.

The nested loop will check if $M1[i].value \leq P[j]$ and $B[M1[i].index].value \geq P[j]$ using an $if\ statement$. In other words, this is used to check if the price of the item is greater than the customers minimum price and lower than their budget. This $if\ statement$ should also check that $M1[i].served$ is false

If this $if\ statement$ passes true, the jewellery, j, will be allocated to $M1[i].index$ and the boolean value $M1[i].served$ will be set to true.

If it is instead false, the nested loop will continue to the next iteration.

Once the nested loop finished, the main loop will continue to the next iteration.

The time complexity of this nested loop is $O(nm)$. To compute the final time complexity, we will sum up all the time complexities of the other actions. However, since $O(nm)$ is the dominant term, the others can be dropped. This will result in a final time complexity of $O(nm)$.

2.3
Consider an algorithm which uses a maximum flow algorithm. The algorithm will implement the Ford Fulkerson method, more specifically, Edmonds-Karp algorithm. This algorithm will run in $O(n^2m)$ time.