DAA Assignment 7

Devam Desai IIT2022035

Q1.

Code:

```
void optimiseJobs(int n, vector<int> &arr, vector<int> &prof,vector<int> &ans){
    map<int,set<int>> m;
    for(int i=0;i<arr.size();i++){
        m[arr[i]].insert(prof[i]);
    }
    for(auto &x:m){
        ans.push_back(*(x.second.rbegin()));
    }
}

void solve(int n){

    vector<int> ans;
    vector<int> arr;
    vector<int> prof;
    int maxi = 1e6;
    for (int i = 0;i < n;i++){
        arr.push_back(rand()%(maxi));
        prof.push_back(rand()%((int)1e4));
    }
    auto start = high_resolution_clock::now();
    optimiseJobs(n,arr,prof,ans);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << (duration).count() << endl;
}
```

Analysis:

The optimiseJobs function solves a simplified version of the job scheduling problem, where each job has an arrival time, a profit, and the goal is to select the jobs with the maximum profit while respecting their arrival times.

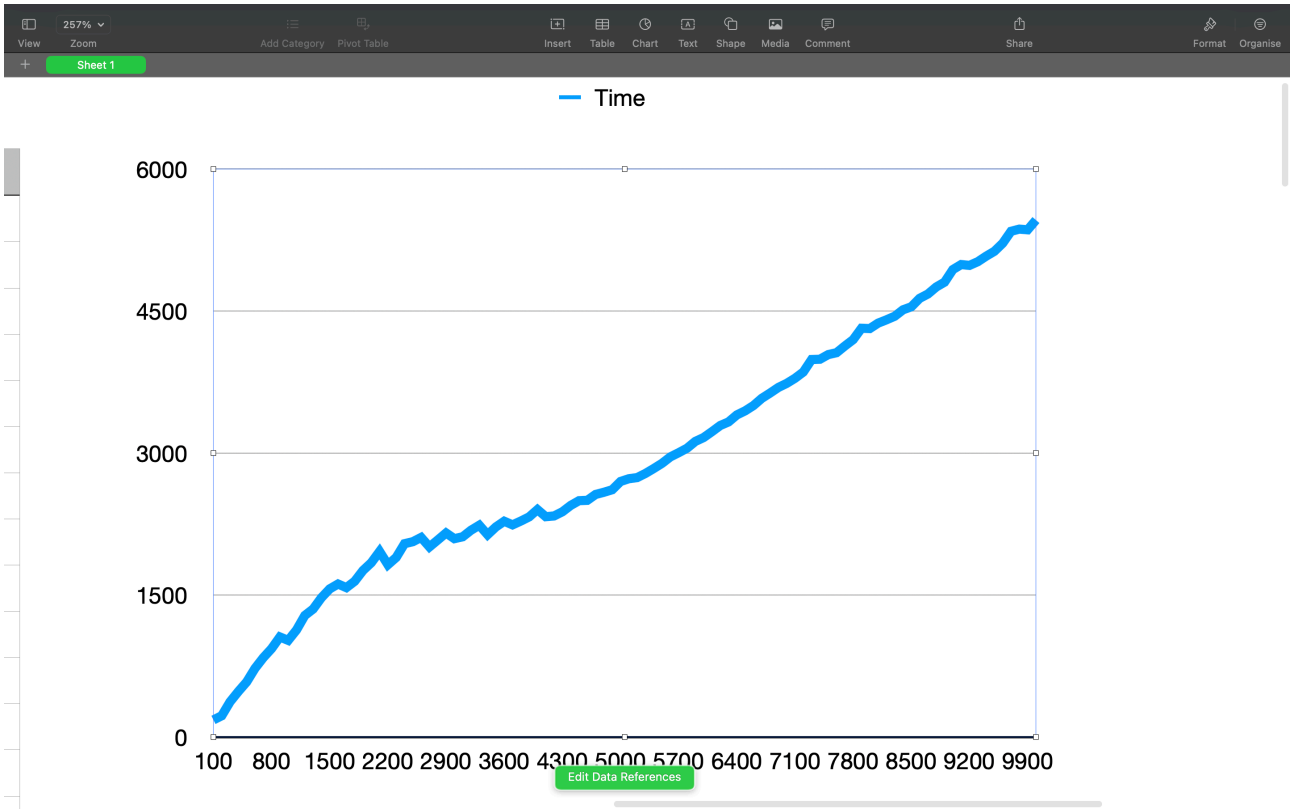
Here's a breakdown of how the function works:

- Data Structure:
  - The function uses a map<int, set<int>> m to store jobs based on their arrival times. The map is ordered by the arrival times, and for each arrival time, there is a set of profits associated with it.
- Insertion into the Map:
  - The function iterates through the input vectors arr (arrival times) and prof (profits).
  - For each job, it inserts the profit into the set corresponding to its arrival time in the map.
- Selecting the Maximum Profit:
  - After populating the map, the function iterates through the map.
  - For each arrival time, it selects the maximum profit from the associated set and adds it to the ans vector.
- Output:
  - The ans vector will contain the arrival time.

This function essentially prioritizes jobs based on their arrival times and selects the one with the highest profit for each arrival time. It ensures that for a given arrival time, the job with the maximum profit is considered.

Overall, the dominant factor in the time complexity is the insertion into the map. Therefore, the overall time complexity of the optimiseJobs function is O(n log n).

Graph:



Q2.

Code:

```
int fractionalKnapsack(int n, vector<int> &wt, vector<int> &cost, int cap){
    map<double,int> m;
    int ans=0;
    for(int i=0;i<n;i++){
        m[wt[i]/(double)cost[i]]=wt[i];
    }

    for(auto it=m.rbegin();it!=m.rend();it++){
        auto x=*it;
        if(cap==0) break;
        int wi=x.second;
        double ci=x.first;
        if(cap<wi){
            cap=0;
            ans+=ci*(double)cap;
        }
        else{
```

```
            ans+=ci*(double)wi;
            cap-=wi;
        }
    }
    return ans;
}

void solve(int n){
    // cout << n << endl;

    vector<int> arr;
    vector<int> wt;
    int maxi = 1e3;
    for (int i = 0;i < n;i++){
        arr.push_back(rand()%(maxi));
        wt.push_back(rand()%((int)1e3));
    }
    int cap=n*10;
    auto start = high_resolution_clock::now();
    int ans=fractionalKnapsack(n,arr,wt,cap);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << (duration).count() << endl;
}
```

Analysis:

Time Complexity Analysis:

- Insertion into the Map:
  - For each item, the function inserts a key-value pair into the map. The insertion operation in a map takes O(log n) time, where n is the number of items.
- Total time complexity for insertion: O(n log n)
- Iteration through the Map:
  - The function then iterates through the map, which takes O(n) time, where n is the number of items.
- Total time complexity for iteration: O(n)

The dominant factor in the time complexity is the insertion into the map, and overall, the time complexity of the fractionalKnapsack function is **O(n log n).**

Fractional Knapsack Problem:

The function implements the fractional knapsack problem solution using a greedy approach. Here's how it works:

- Sorting by Value-to-Weight Ratio:
  - The function uses a map to store the value-to-weight ratios of items as keys and the corresponding weights as values.
  - This automatically sorts the items in descending order based on their value-to-weight ratios.
- Greedy Selection:
  - The function then iterates through the sorted map in reverse order (highest ratio first).
  - It selects items greedily, considering the maximum possible weight until the knapsack capacity is exhausted.
- Calculation of Total Value:
  - For each selected item, it calculates the contribution to the total value based on the available capacity and the item's weight.
- Output:
  - The function returns the total value obtained from the fractional knapsack.

In summary, the function optimally selects items based on their value-to-weight ratios, prioritizing items with higher ratios. This greedy approach ensures that the knapsack's capacity is used

efficiently, maximizing the total value that can be accommodated. The time complexity of the function is reasonable for moderate-sized inputs, making it suitable for solving the fractional knapsack problem.

Graph: