ALGORITHM DESIGN, PSEUDOCODE, PROOF OF CORRECTNESS & ANALYSIS OF RUNNING TIME

VIRAT ARUN JOSHI (A20417850) SUNNY MODI (A20424212)

DESIGN

The problem statement provides us with the following parameters :
Number of jobs (n), number of machines (m), start time of the job, finish time of the job
We use the greedy approach of "earliest finish time first" to schedule the jobs. All jobs are sorted in ascending order of their finish times. After that, the job with the earliest finish time is scheduled for the first machine. The next job is scheduled if compatible with the finish time of the previous job. This is repeated until no more jobs can be scheduled for the first machine. In the same way, we schedule the remaining jobs for all subsequent machines. As a result, each machine is assigned the maximum number of jobs possible.
If the jobs have common finish times, the tie is arbitrarily broken in the sorting phase. This because changing the order of these jobs will no effect on the optimality of the scheduling.
The data structures used in the algorithm are arrays and dictionaries. Dictionaries are used since we can store the job number as the key and the corresponding start and finish times as values. We also use try-catch blocks in our execution as a way of handling exceptions
The finish times are sorted using python's inbuilt function "*sorted()*" which implements a hybrid of merge sort and insertion sort. The algorithm finds subsets of the data that are already ordered and uses the subsets to sort the data more efficiently. This is done by merging an identified subset, called a run, with existing runs until certain criteria are fulfilled. The worst-case runtime for sorting is O(n logn). The total runtime of the algorithm is *m x nlogn* which we will prove later.

PSEUDOCODE

We now provide the pseudo code of our algorithm.

main() Function

```
1. Read input file and extract the number of jobs n and number of
   machines m.
2. Store start and finish times of all n jobs.
3. Sort the finish times to obtain the earliest finish time.
4. Initialize arrays to store the scheduled jobs and the remaining
   jobs along with a counter to maintain the scheduled jobs count.
5. For each machine m :
     a. If there are jobs remaining:
          i.   Call the schedule() function.
         ii.   Store the unscheduled jobs in an array.
        iii.   Store the scheduled jobs in an array.
6. Call the output function to write the results to a text file
```

schedule() function

```
1. For each job :
     a. If the start time of current job ≥ finish time of the job
        that was last scheduled :
          i.   Schedule the job
```

```
      b. Else:
           i.   Add job to list of remaining jobs
   2. Return scheduled jobs and remaining jobs
```

`output() function`

```
   1. Write the count of total jobs scheduled to the output file
   2. For each machine m:
        a. Write the jobs scheduled in order.

Call the main() function.
```

<u>PROOF OF CORRECTNESS</u>

We know that the output will never contain conflicting schedule. This is because the start time of each job will always be greater than the finish time of the previously scheduled job.

To prove that our greedy approach is optimal, we will consider that there is an alternate optimal solution and compare it with the greedy approach. If both the solutions are equal, we will know that the greedy approach is optimal. Else, we consider the instance at which the solutions are different and prove that the greedy approach is better by induction.

First, we prove this for the schedule of a single machine.

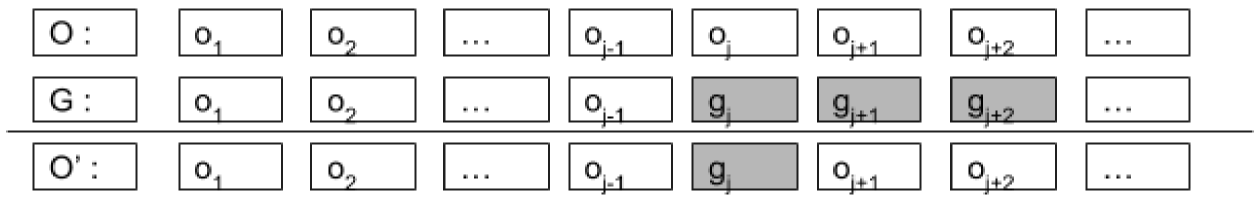
Claim: The Greedy approach of using the earliest finish time first is optimal

Proof:

Let us assume that our greedy solution has a schedule of jobs G = $(g_1, g_2, \dots g_p)$ and the optimal solution has a schedule O = $(o_1,o_2, \dots o_q)$.

If O is optimal it will have the same number of jobs as G and the solutions will differ at some job j such that O = $(o_1,o_2, \dots o_{j-1}, o_j, \dots o_q)$ and G = $(o_1, o_2, \dots o_{j-1}, g_j, \dots g_p)$ where $o_j \neq g_j$ and $p \geq j$. If $p < j$ G would have more jobs than O thus contradicting the statement that O is an optimal solution.

The greedy approach will select the job with the earliest finish time such that it does not conflict with previously scheduled jobs. Thus g_j does not conflict with any of the previous jobs and finishes at the same time as o_j.



If we consider a modified greedier schedule by replacing o_j with g_j in O as shown above, it will be a valid schedule because both jobs will finish at the same time and thus there will be no conflict. The schedule O' will be now $(o_1, o_2, \dots o_{j-1}, g_j, o_{j+1}, \dots o_q)$. This schedule will have the same number of jobs and is at least as good as our previous optimal solution. We can repeat this step for all the jobs and convert O into G keeping the number of jobs constant. Hence, we can say that our greedy solution G is optimal.

We proved that our algorithm is able to give an optimal solution for a single machine. The only modification in the implementation for m machines is that after scheduling the jobs for the first machine, we schedule the remaining jobs for the next machine and so on. Thus at each machine, our greedy solution will be optimal thus giving us an optimal solution in general.

ANALYSIS OF RUNNING TIME

To analyse running time we calculate the running time of all operations. Let n be the number of jobs and m be the number of machines.

1. initialization of data structures: *O(1)* for each operation
2. Line-wise I/O operations: *O(n)*
3. Iterating over each job to store start and finish times: *O(n)*
4. Sorting finish time for each jobs using hybrid sort (Tim Sort): *O(n*logn)*
5. Calling schedule function for m machines and n jobs: *O(m * n)*
6. Calling output function: *O(n)*

Here we know that the cost of running the schedule and output function is O(n) since we iterate n times in a for loop. All other operations are *O(1)*. A nested for loop in the main() function iterates through *m* machines for a total of *n* jobs.
For large values of m and n, we can ignore the *O(1)* costs. Thus the total running time can be given as "*n + n + nlogn + m*n + n*" from the above steps. It can be written as *O(n * (logn + m + 3))* which gives us ***O(nlogn + n*m)*** or ***O(n*(logn + m))*** in the worst case. Thus our algorithm runs in polynomial time as required.
(We should note that Tim-sort operates in linear time for most real-world data giving us a running time of O(n*m) for small inputs.)

CONCLUSION

We implement a polynomial time interval scheduling algorithm with the approach of "*earliest finish time first".* We also prove that this greedy approach is the optimal solution for our problem statement and then analyse the run-time to see that the requirements are met.