# Documentation - Job Scheduling

## Data Structures Used:

1. **'jd' struct** :- Structure representing the job
2. **'aspq' priority queue**:- Current global waiting queue.
3. **'ospq' priority queue** :- Current list of waiting queues for each origin
4. **'cpupq' priority queue** :- Priority queue signifying the processes that are currently in CPU(prioritized according to the ending time)
5. **'snap' struct** :- Structure representing a particular transaction of adding or removing a job from waiting priority queue.
6. **'snap_vec' vector** :- vector of all the transactions that have happened till now
7. **'qkps'** :- Array of copies of global waiting priority queue at different time intervals currently (after 1000 transactions ~ intended to be sqrt(n))
8. **'qomps'** :- Array of copies of waiting priority queues for each origin at different time intervals (after 1000 transactions ~ intended to be sqrt(n))
9. **'snap_pos' array** :- Array to map each copy of waiting priority queue in 'qkps' to the corresponding position in 'snap_vec' (snap vector of transactions)

## Why We Choose What We Choose?

1. **'jd' struct** - We could have taken individual array for each data information but with that managing individual would become cumbersome. Thus, we made a structure of all information related to a job.
2. **Priority queue** - We need to be able to identify the maximum priority process at any given moment of time. Further, priority queue enables us to insert and delete element in time complexity O(logn).
3. **Snaps** - Query can belong to a moment in past. Thus, we would need to simulate the transactions that happened in the past. But, due to memory constraint we can't store the state of queue at each instance. Thus, we store only the transactions from one state to next state.
4. **'qkps' and 'qomps' Array** - In order to query, we would need to simulate transactions, but simulating transactions each time would be of complexity O(n). Thus we would compromise on memory and store the state of ~sqrt(n) (Currently taken 1000) instances. Thus, the time-complexity would reduce to O(sqrt(n)).

## Algorithm:

**Case 1: Job is added**
We add the job in the global priority queue and the priority queue of its corresponding origin.
We also add the job transaction in the ‘snap_vec’ to be used by ‘query’ later. If the ‘cnt’ reaches 1000 we store the copy of current priority queues in ‘qkps’ and ‘qomps’ and resent the ‘cnt’.
Time Complexity: log(n)

**Case 2: assign <timestamp> <k>**

First of all we complete all the tasks whose end time are less than ‘timestamp’ from ‘cpupq’ i.e. they have been completed by now.
Now, we remove the min(k,f,s) highest priority jobs from waiting queue and add them in ‘cpupq’.
We also add each of the ‘remove’ transaction into the ‘snap’ vector.
Time Complexity: log(n)

**Case 3: query <timestamp> <k> or query <timestamp> <origin>**

As we have stored the snaps of priority queue at different time intervals(after each 1000 transactions), we will go to that copy of priority queue which is just before the time specified in the query and from there we again execute all the needed transactions(using snap_vec) to get the required result.
The complexity is (n/1000)*1000*log(n)  (no. of priority queue snaps * transactions b/w two snaps) where 'n' is total number of transactions.

Case 3.1: query <timestamp> <k>
Here we take the required copy of global priority queue from ‘qkps’ and do the required operations to reach the exact priority queue that existed at ‘timestamp’.
Now we print the top ‘k’ jobs of that priority queue.

Case 3.2: query <timestamp> <origin>
Here we take the required copy of priority queue from ‘qops’ and do the required operations to reach the exact priority queue that existed at ‘timestamp’ for the particular ‘origin’.