Q1 Interval Scheduling on \( m \) Machines

Let us recall the interval scheduling problem from class. We are given \( n \) jobs, where each job \( I_j = [s_j, f_j) \) is an interval. Two jobs are compatible if their intervals have empty intersection. In class, we saw how to schedule a maximum number of mutually compatible jobs on one machine: consider the jobs one-by-one in an increasing order of their finish time (EFT), and greedily pick the job being considered if it is compatible with the ones picked so far.

Now, suppose that we have \( m \) machines available instead of just one. A feasible schedule can be thought of as a mapping \( \sigma : \{1, \ldots, n\} \rightarrow \{0, 1, \ldots, m\} \), where job \( I_j \) is assigned to machine \( k \) if \( \sigma(j) = k > 0 \), and unassigned if \( \sigma(j) = 0 \). Jobs scheduled on any single machine must still be mutually compatible. Subject to that, we want to maximize the number of jobs scheduled, i.e., \( |\{j \in \{1, \ldots, n\} : \sigma(j) > 0\}| \).

(a) Consider the following Earliest Start Time (EST) algorithm.

\begin{algorithm}
\caption{m-ISP-EST}
1 Sort the jobs in an increasing order of their start time so that \( s_1 \leq \ldots \leq s_n \)
2 for \( j = 1, \ldots, n \) do
3 \hspace{1em} if there is a machine \( k \) such that all jobs assigned to it are compatible with \( I_j \) then
4 \hspace{2em} Assign job \( I_j \) to any such machine \( k \)
5 \hspace{1em} else
6 \hspace{2em} Let job \( I_j \) remain unscheduled
7 \hspace{1em} end
8 end
\end{algorithm}

Consider the following attempt to prove optimality of this algorithm.

1. Consider any job \( I_j \) that the algorithm does not schedule.
2. At that point, each machine must have a job scheduled that conflicts with \( I_j \).
3. As seen in class, all \( m \) conflicting jobs must have start time earlier than \( s_j \) (due to the EST order) and finish time after \( s_j \) (since they conflict with \( I_j \)).
4. Hence, there are at least \( m + 1 \) jobs that all contain time \( s_j \) in their intervals.
5. Since there are only \( m \) machines, even the optimal algorithm must drop at least one of them.
6. The optimal algorithm drops a job for every job dropped by our EST algorithm. Hence, our EST algorithm schedules the maximum number of jobs.

Which step(s) of the argument above are flawed, and why?

(Solution to (a)) The sixth step is flawed. We would need to show that for every job dropped by the EST algorithm, the optimal solution drops a distinct job. Otherwise, the optimal solution may drop one job that conflicts with two or more jobs dropped by the EST algorithm, thus leading to more jobs scheduled.
Next, consider the Earliest Finish Time (EFT) algorithm.

**Algorithm 2: m-ISP-EFT**

1. Sort the jobs in an increasing order of their finish time so that \( f_1 \leq \ldots \leq f_n \)
2. for \( j = 1, \ldots, n \) do
   3. if there is a machine \( k \) such that all jobs assigned to it are compatible with \( I_j \) then
   4. Assign job \( I_j \) to any such machine \( k \)
   5. else
   6. Let job \( I_j \) remain unscheduled
   7. end
8. end

Prove that this algorithm does not always yield an optimal solution by producing a counterexample.

**(Solution to (b))** Consider an instance with \( m = 2 \) and four jobs: \( I_1 = [1, 3) \), \( I_2 = [2, 4) \), \( I_3 = [4, 5) \), and \( I_4 = [3, 6) \). Note that these are sorted by EFT. The EFT algorithm schedules the first two jobs on two different machines. In the third round, it has a choice to schedule \( I_3 \) on either machine. If it ends up scheduling it on the same machine as \( I_1 \), then \( I_4 \) will have to be dropped. This would be suboptimal because scheduling all four jobs — \( I_2 \) and \( I_3 \) on one machine and \( I_1 \) and \( I_4 \) on the other — is feasible.

(c) Finally, consider the EFT algorithm, but with a smart tie-breaking in case there are multiple machines which can accommodate job \( I_j \).

**Algorithm 3: m-ISP-EFT-Best-Fit**

1. Sort the jobs in an increasing order of their finish time so that \( f_1 \leq \ldots \leq f_n \)
2. for \( j = 1, \ldots, n \) do
3. Let \( S \) be the set of machines such that all jobs assigned to them are compatible with \( I_j \)
4. if \( S \neq \emptyset \) then
5. For each machine \( k \in S \), let \( e_k \) be the greatest finish time of any job assigned to it so far
6. Assign job \( I_j \) to machine \( k \in S \) with the greatest \( e_k \)
7. else
8. Let job \( I_j \) remain unscheduled
9. end
10. end

Prove that this algorithm always yields an optimal solution. Design an efficient implementation and analyze its worst-case running time.

**(c) Correctness:** Let \( G_i \) be the greedy schedule after the first \( i \) intervals have been processed. We prove by induction on \( i \) that for each \( i \in \{0, 1, \ldots, n\} \), there exists an optimal schedule \( OPT_i \) that extends \( G_i \) — that is, matches \( G_i \) on the scheduling of jobs \( I_1 \) through \( I_i \). This shows that \( G_n \) is optimal.

The base case of \( i = 0 \) trivial: since \( G_0 = \emptyset \), any optimal solution \( OPT_0 \) extends it. Suppose the inductive hypothesis holds for \( i - 1 \). We prove it for \( i \) by considering the following cases:
• \(I_i\) is dropped in \(G_{i-1}\). This implies that no extension of \(G_{i-1}\) can schedule \(I_i\) on any machine. In particular, \(OPT_{i-1}\) cannot schedule \(I_i\) either. Thus, \(OPT_{i-1}\) extends \(G_i\) as well.

• \(I_i\) is scheduled on the same machine in \(G_i\) and in \(OPT_{i-1}\). Then, clearly \(OPT_{i-1}\) again extends \(G_i\).

• \(I_i\) is scheduled on some machine \(k\) in \(G_i\), but is dropped in \(OPT_{i-1}\). Let \(I_j\) (with smallest \(j > i\)) be the next interval scheduled on machine \(k\) under \(OPT_{i-1}\). Note that such an interval must exist, otherwise, by scheduling \(I_i\) on machine \(k\) under \(OPT_{i-1}\), we could schedule one extra job, which would contradict optimality of \(OPT_{i-1}\). Define \(OPT_i\) by replacing the interval \(I_j\) with \(I_i\) in \(OPT_{i-1}\). To see why this is feasible, suppose \(e_k\) is the latest finish time of any job scheduled on machine \(k\) before job \(i\). This quantity is the same under \(G_{i-1}\) and \(OPT_{i-1}\) because the latter extends the former. Because \(G_i\) extends \(G_{i-1}\) by scheduling \(I_i\) on machine \(k\), we must have \(e_k \leq s_i\), and because \(OPT_{i-1}\) extends \(G_{i-1}\) by scheduling job \(j\) next on machine \(k\), the machine is available until \(f_j\) after removing job \(I_j\). Since \(j > i\), we have \(f_j \geq f_i\). So \(I_i\) can be scheduled on machine \(k\). Note that replacing job \(j\) with \(i\) keeps the schedule optimal.

• \(I_i\) is scheduled on some machine \(k\) in \(G_i\), and some different machine \(k'\) in \(OPT_i\). Construct \(OPT_i\) by starting with \(OPT_{i-1}\), and swapping the schedules of machines \(k\) and \(k'\) after time \(s_i\). That is, for all jobs \(j \geq i\), if \(I_j\) is scheduled on machine \(k\) (resp. \(k'\)), then we move it to machine \(k'\) (resp. \(k\)).

For an efficient implementation, we note that we only need the latest finish time on each machine, which we can keep track of easily because it is always the finish time of the last job scheduled on the machine under the EFT order. The key step is to search for the maximum latest finish time that does not exceed the start time of the job under consideration. The following implementation uses binary search trees (BST) to do this efficiently.

\begin{algorithm}
\caption{m-ISP-EFT-Best-Fit}
\begin{algorithmic}
\State Sort the jobs in an increasing order of their finish time so that \(f_1 \leq \ldots \leq f_n\)
\State Create a BST of the \(m\) machines using key \(e_k \leftarrow 0\) for each machine \(k\)
\For {\(j = 1, \ldots, n\)}
\State Search the BST for the greatest key that is less than or equal to \(s_i\)
\If {there is no such key}
\State Let job \(I_j\) remain unscheduled
\Else
\State If the key corresponds to machine \(k\), schedule \(I_j\) on \(k\) and update its key \(e_k \leftarrow f_j\)
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

This algorithm spends \(O(\log m)\) time in each iteration of the for loop, so its worst-case running time is \(O(n \log m + m)\), where the latter term is for creating the BST of size \(m\) in the first place.
**Q2 Cops and Robbers**

You are given an array $A[1, \ldots, n]$ with the following specifications:

- Each element $A[i]$ is either a cop ('c') or a robber ('r').
- A cop can catch any robber who is within a distance of $K$ units from the cop, but each cop can catch at most one robber.

Write an algorithm to find the maximum number of robbers that can be caught.

**Solution to Q2**

**Algorithm 5: Cops-and-Robbers**

1. for $c = 1, \ldots, n$ do
2. 
3. if $A[c]$ is a cop then
4.  
5.  Find the smallest index $r$ in $[c-K, c+K]$ such that $A[r]$ is a robber who is not currently assigned to any cop
6.  
8. end
9. end

**Correctness:** Suppose for contradiction that this algorithm is not optimal. Let $G$ denote the greedy assignment it produces. Let $OPT$ be an optimal solution that matches $G$ for as many iterations as possible. That is, for some $k$, $OPT$ assigns the first $k$ cops exactly the same way as $G$ does, and there is no optimal solution which matches $G$ on the first $k + 1$ cops. Because $G$ is not optimal, $k$ is less than the number of cops.

We derive a contradiction by proving that there is an optimal solution $OPT'$ which matches $G$ on the first $k + 1$ cops. Let $c$ denote the index of the $k + 1$th cop. We consider the following cases:

1. Suppose $G$ leaves cop $A[c]$ unassigned, while $OPT$ assigns cop $A[c]$ to robber $A[r]$. In this case, $r \in [c-K, c+K]$. However, since $OPT$ matches $G$ on the first $k$ cops, $A[r]$ must not be assigned to the first $k$ cops under $G$. This is a contradiction as the greedy algorithm would not have left cop $A[c]$ be unassigned in that case.


3. Suppose $G$ assigns cop $A[c]$ to some robber $A[r]$, while $OPT$ assigns cop $A[c]$ to some other robber $A[r']$. If $A[r]$ is not assigned to any cop under $OPT$, then we can create $OPT'$ by switching the assignment of $A[c]$ from $A[r']$ to $A[r]$. If $A[r]$ is assigned to some cop $A[c']$ under $OPT$, then we create $OPT'$ by switching the assignments of $A[c]$ and $A[c']$, i.e., by assigning $A[c] \rightarrow A[r]$ and $A[c'] \rightarrow A[r']$. For this to work, we need to argue that $A[c'] \rightarrow A[r']$ is a valid assignment, i.e., that $r' \in [c' - K, c' + K]$. 


Because $OPT$ and $G$ match on the assignment of the first $k$ cops, we have $c' > c$. Further, because $G$ greedily assigns cop $A[c]$ to the first feasible robber not assigned to any previous cops, we have $r' > r$. Now, because $OPT$ assigns $A[c']$ to $A[r]$, we have $c' - K \leq r \leq r'$. On the other hand, because $OPT$ also assigns $A[c]$ to $A[r']$, we also have $r' \leq c + K \leq c' + K$. This completes the proof of validity of $A[c'] \rightarrow A[r']$ assignment.

**Running Time:** The above algorithm clearly runs in time $O(nK)$ because for each of $O(n)$ cops, it searches for the first available feasible robber in $O(K)$ time. However, the running time can be reduced to $O(n)$ by keeping track of two indices $c$ and $r$ for the next available cop and robber, and increasing both when a match is made or the minimum when they are too far, as the algorithm below shows.

**Algorithm 6: Cops-and-Robbers**

1. Initialize $c$ and $r$ to the indices of the first cop and the first robber, respectively
2. while $c$ and $r$ are not null do
   3. if $|c - r| \leq K$ then
      4. Assign cop $c$ to catch robber $r$
      5. Increment both $c$ and $r$ to the indices of the next cop and robber, respectively
   6. else if $c < r$ then
      7. Increment $c$ to the index of the next cop
   8. else
      9. Increment $r$ to the index of the next robber
   10. end
3. end