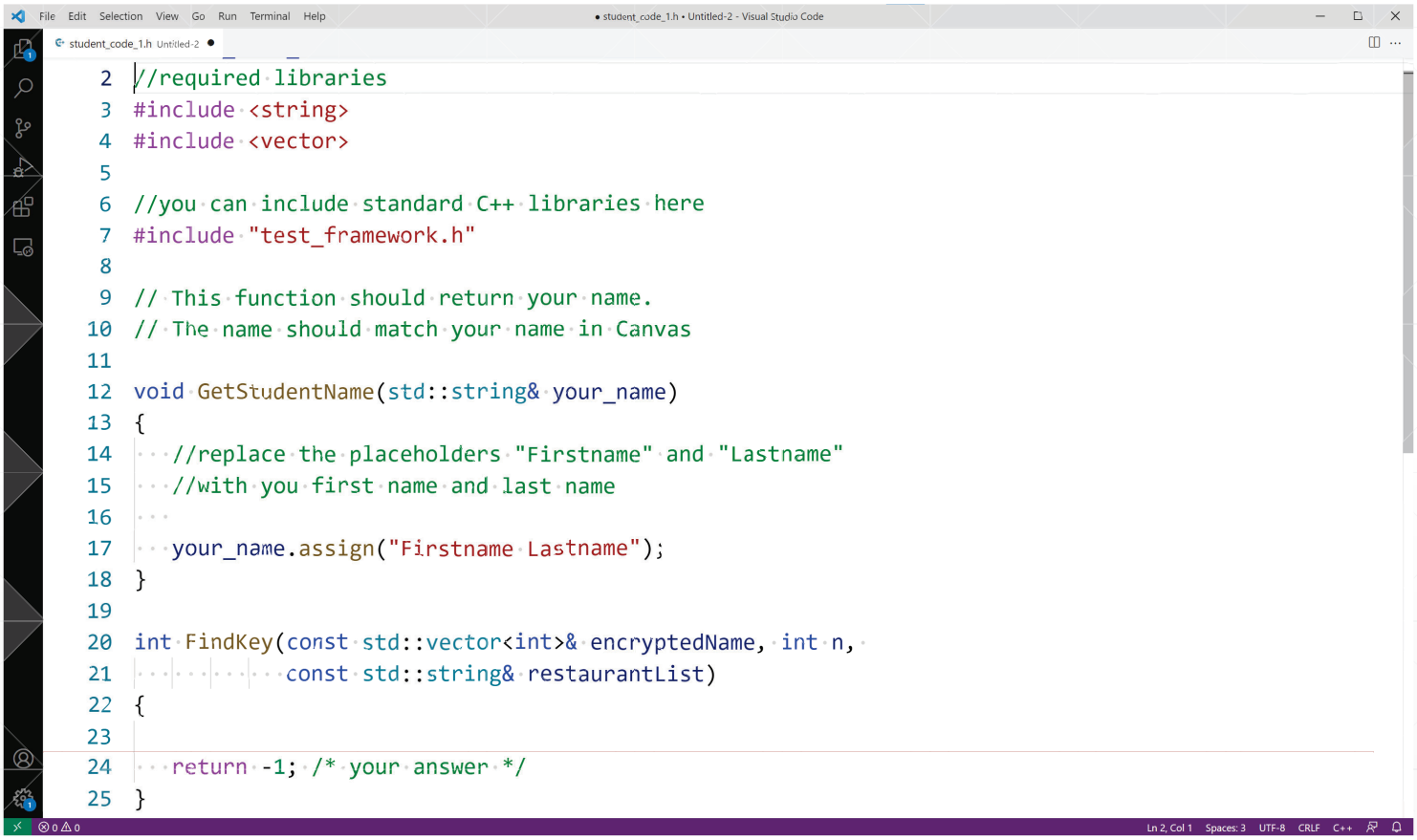


Interval Scheduling

CS 336: Design and Analysis of Algorithms

Table Reservation System

- Restaurant: m tables.
- Receive n requests: $[s(j), f(j)]$
  - $s(j)$ — start time; $f(j)$ — finish time.
- Goal: Accept the maximum # of reservations.

Interval Scheduling

- **Given:** Jobs $j \in \{1, \ldots, n\}$ with start time $s(j)$ and finish time $f(j)$.
- **Goal:** Maximize # scheduled jobs.

requests = jobs; tables = machines

Applications: cloud computing & operating systems.

General Approach

Rank all jobs – *TBD*.
Sort all jobs j according to the rank. Add them to array J.

for every job j in J:
  Reserve a machine for time $[s(j), f(j)]$ for job j if possible;
  Otherwise, discard the job.

How to Rank Jobs?

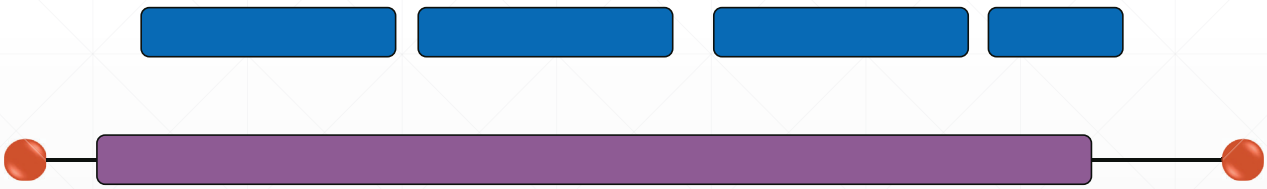
- Consider the case when we have only one machine.
- All jobs have the same value.
- Pick the one with
  - Minimum start time $s(j)$. First come first serve.
  - Minimum processing $f(j) - s(j)$.
  - Minimum finish time $f(j)$.
- Assume that we have only one machine for now.

How to Rank Jobs?

- Minimum start time s_j. First come first serve.

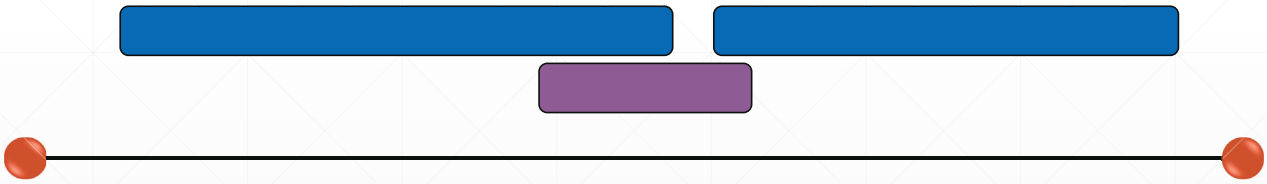
How to Rank Jobs?

- Minimum start time s_j. First come first serve.



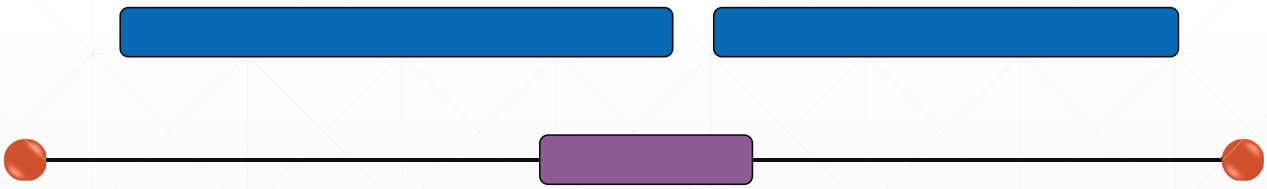
How to Rank Jobs?

- Minimum processing time $f(j) - s(j)$. Shortest jobs first.



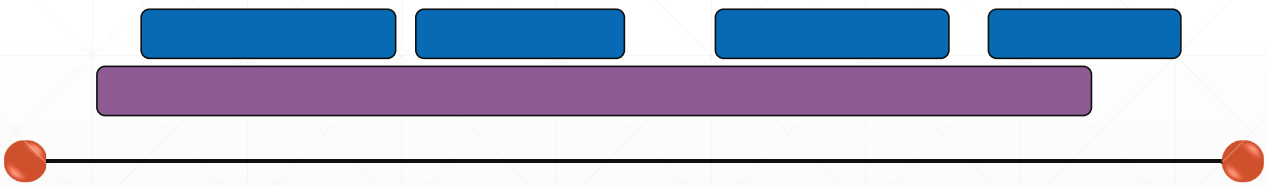
How to Rank Jobs?

- Minimum processing time $f(j) - s(j)$. Shortest jobs first.



Better Idea

- Let's pick jobs that finish first.



Theorem

Let **Alg** be greedy algorithm that first picks jobs that finish first.

- **Alg** always finds the optimal solution.
- The running time of **Alg** is $O(n \log n)$.

Proof:
First consider the case when we have only one machine**.**

Proof:
First consider the case when we have only one machine**.**

Proof:
First consider the case when we have only one machine ($m = 1$**).**

Let

- $\boldsymbol{A}$ be the solution – set of jobs – returned by **Alg**.
- $\boldsymbol{O}$ be the set of jobs in an optimal solution.
- Sort all jobs in $\boldsymbol{A}$ and $\boldsymbol{O}$ by finish time.
- Let $\boldsymbol{A} = \{a_1, \dots, a_k\}$ and $\boldsymbol{O} = \{o_1, \dots, o_{k'}\}$.

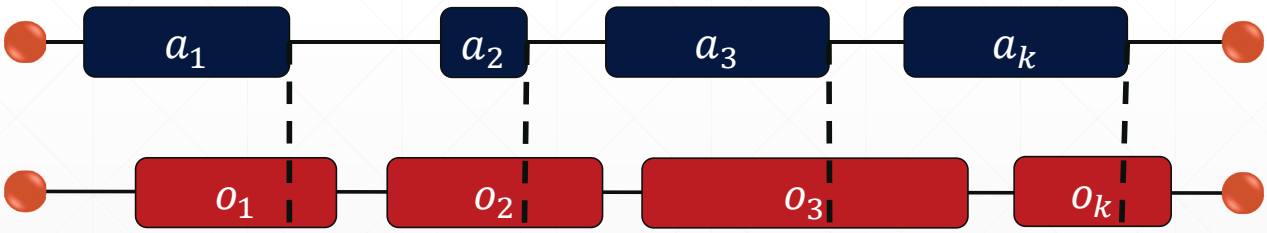
Claim: $f(a_i) \leq f(o_i)$ for all $i \leq k$

- Algorithm stays ahead of OPT, i.e., a_i finishes earlier than o_i for all i.

Two solutions

A – solution returned by the algorithm Alg.

O – optimal solution.



Inductive hypothesis: $f(a_i) \leq f(o_i)$

- $A = \{a_1, \ldots, a_k\}$ – solution returned by **Alg**.
- $O = \{o_1, \ldots, o_{k'}\}$ — optimal solution

Proof by induction.

Base case: $i = 1$. Need to show $f(a_1) \leq f(o_1)$. Why?

Inductive hypothesis: $f(a_i) \leq f(o_i)$

- $\boldsymbol{A} = \{a_1, \dots, a_k\}$ – solution returned by **Alg**.
- $\boldsymbol{O} = \{o_1, \dots, o_{k'}\}$ — optimal solution

Proof by induction.

Base case: $i = 1$. Need to show $f(a_1) \leq f(o_1)$. Why?

Because at the first step **Alg** picks a job j with minimal $f(j)$.

Inductive hypothesis: $f(a_i) \leq f(o_i)$

- $\boldsymbol{A} = \{a_1, \dots, a_k\}$ – solution returned by **Alg**.
- $\boldsymbol{O} = \{o_1, \dots, o_{k'}\}$ — optimal solution

Proof by induction.

Inductive step: Assume $f(a_i) \leq f(o_i)$. Why $f(a_{i+1}) \leq f(o_{i+1})$?

Inductive hypothesis: $f(a_i) \leq f(o_i)$

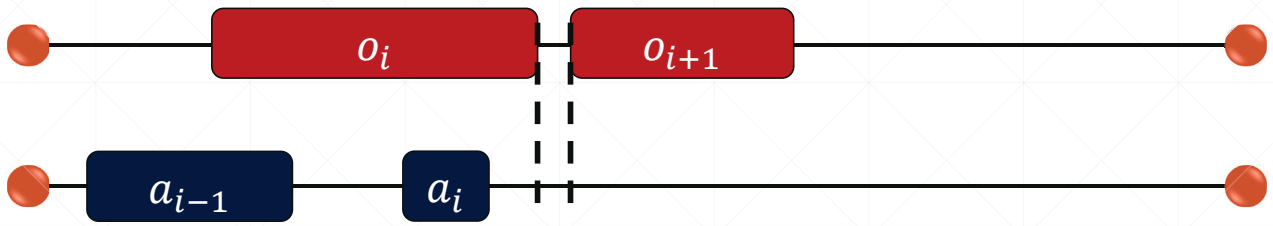
- $\boldsymbol{A} = \{a_1, \ldots, a_k\}$ – solution returned by **Alg**.
- $\boldsymbol{O} = \{o_1, \ldots, o_{k'}\}$ — optimal solution

Proof by induction.

Inductive step: Assume $f(a_i) \leq f(o_i)$. Why $f(a_{i+1}) \leq f(o_{i+1})$?

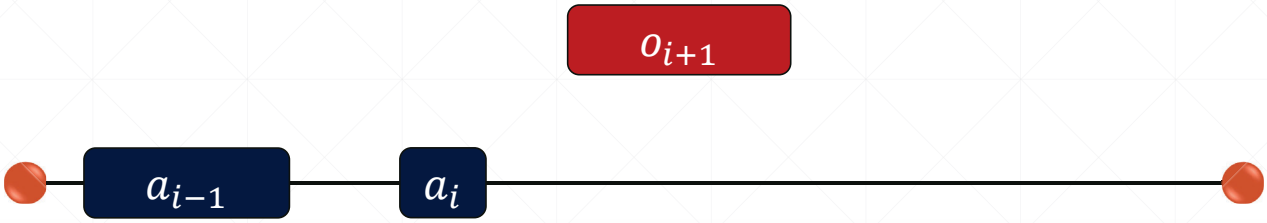
Short Answer: Alg could pick job o_{i+1} instead of a_{i+1} at step $(i+1)$, but chose a_{i+1}. Hence, $f(a_{i+1}) \leq f(o_{i+1})$.

At Step $i+1$:



- Job $\boldsymbol{o_{i+1}}$ starts after job $\boldsymbol{o_i}$ finishes: $s(o_{i+1}) \geq f(o_i)$.
- Job $\boldsymbol{o_i}$ finishes after $\boldsymbol{a_i}$ finishes (by induction hypothesis): $f(o_i) \geq f(a_i)$.
- Hence $\boldsymbol{o_{i+1}}$ starts after $\boldsymbol{a_i}$ finishes: $s(o_{i+1}) \geq f(a_i)$.

At Step $i + 1$:

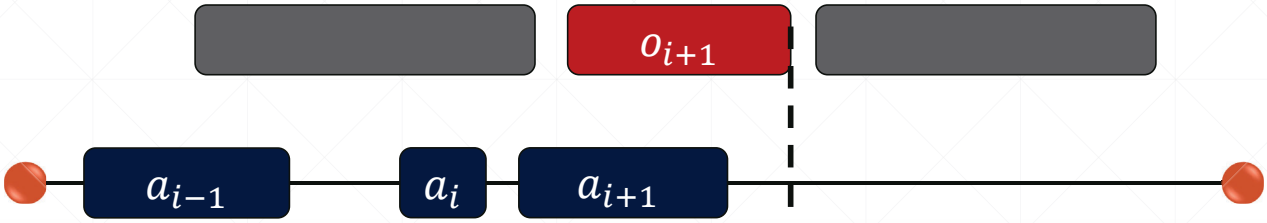


- Job o_{i+1} starts after job o_i finishes: $s(o_{i+1}) \geq f(o_i)$.
- Job o_i finishes after a_i finishes (by induction hypothesis): $f(o_i) \geq f(a_i)$.
- Hence o_{i+1} starts after a_i finishes: $s(o_{i+1}) \geq f(a_i)$.

At Step $i + 1$:

- Job o_{i+1} starts after job o_i finishes: $s(o_{i+1}) \geq f(o_i)$.
- Job o_i finishes after a_i finishes (by induction hypothesis): $f(o_i) \geq f(a_i)$.
- Hence o_{i+1} starts after a_i finishes: $s(o_{i+1}) \geq f(a_i)$.

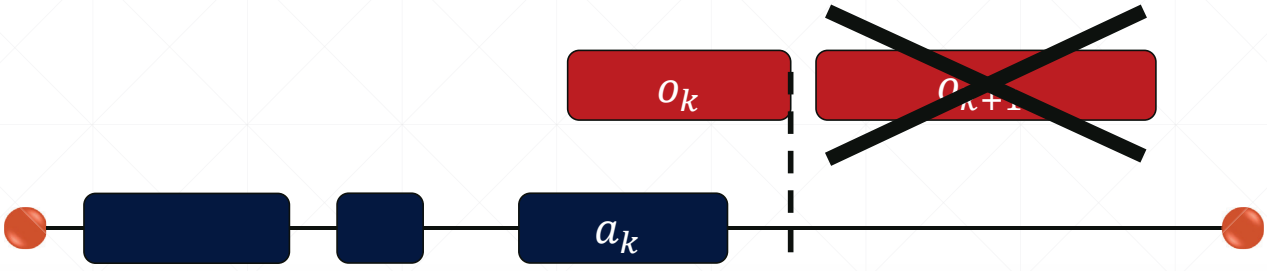
At Step $i + 1$:



- Job o_{i+1} starts after job o_i finishes: $s(o_{i+1}) \geq f(o_i)$.
- Job o_i finishes after a_i finishes (by induction hypothesis): $f(o_i) \geq f(a_i)$.
- Hence o_{i+1} starts after a_i finishes: $s(o_{i+1}) \geq f(a_i)$.

■

After Step k:



- If O had $(k + 1)$ or more jobs, then **Alg** would add job o_{k+1} to its schedule.

■

Running Time

Insert all jobs j in an array J. Sort L by $f(j)$.

for all jobs j in J:
Reserve a machine for time $[s(j), f(j)]$ for job j if possible;
Otherwise, discard the job.

Time: $O(n \log n)$ operations to sort all jobs.

Total running time: $O(n \log n)$.

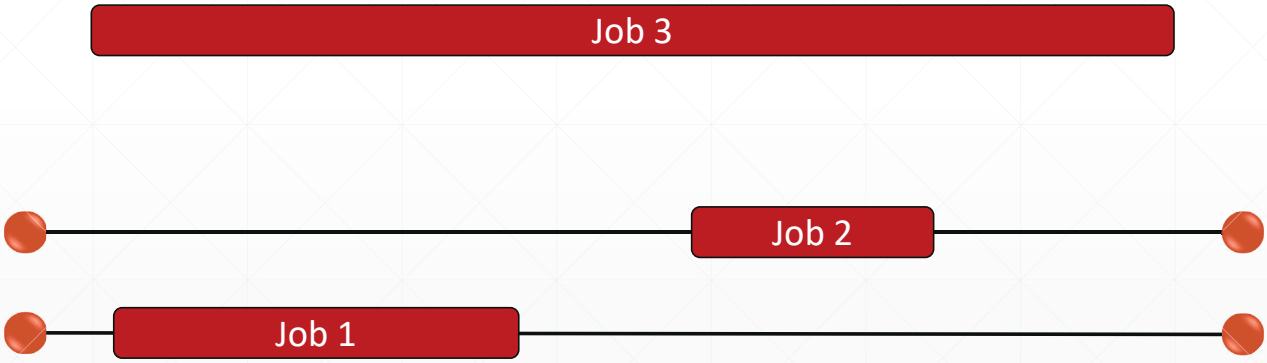
Multiple Machines

Can we use greedy for more than two machines?

Can we use greedy for multiple machines?

Can we use greedy for multiple machines?

Can we use greedy for multiple machines?



Conclusions

- Learned about a greedy algorithm for scheduling.
- Caution: Some scheduling problems are much harder. Greedy algorithms don't always work!

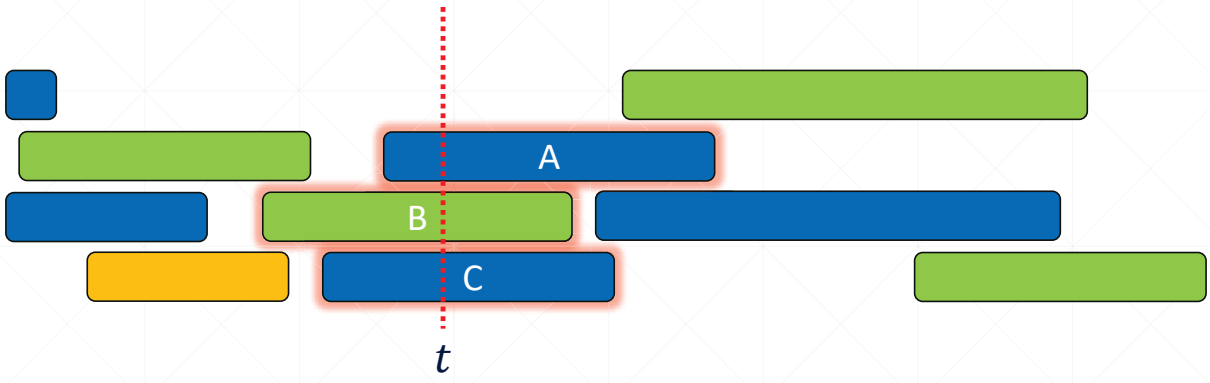
Machine Minimization

CS 336: Design and Analysis of Algorithms

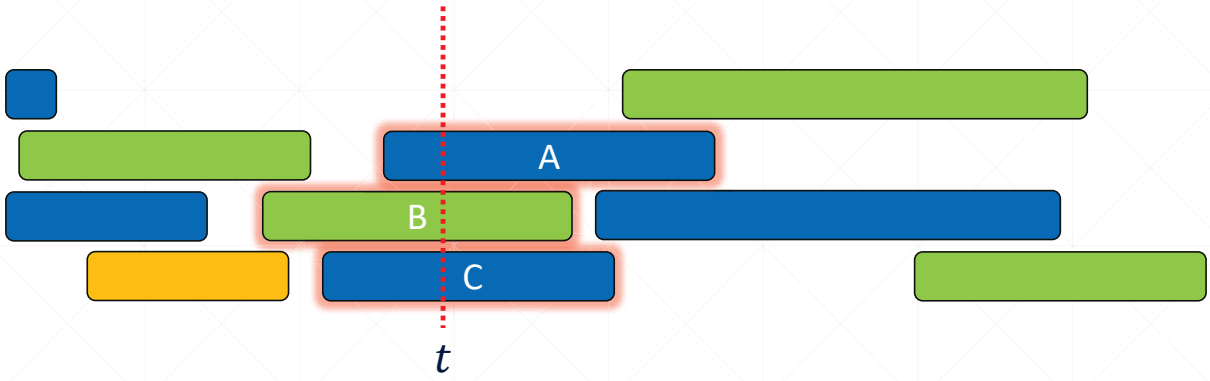
Minimizing # Machines

- Jobs: $1, \ldots, n$
- Job j must start at time $s(j)$ and finish at time $f(j)$.

  $s(j)$ — start time; $f(j)$ — finish time.
- Goal: Minimize # machines used.



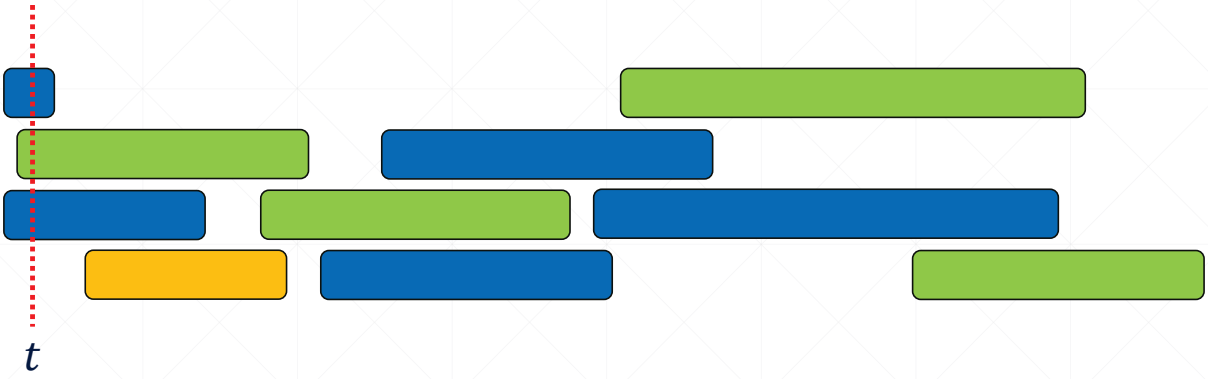
Job j is active at time t if $s(j) \leq t \leq f(j)$.



Job j is active at time t if $s(j) \leq t \leq f(j)$.

Let m_t be # active jobs at time t.

Claim: If at time t, m_t jobs are active, then $OPT \geq m_t$;
here OPT is the optimal # machines.



Job j is active at time t if $s(j) \leq t \leq f(j)$.

for all t:

  m_t = # active jobs at time t

return $\max_t m_t$

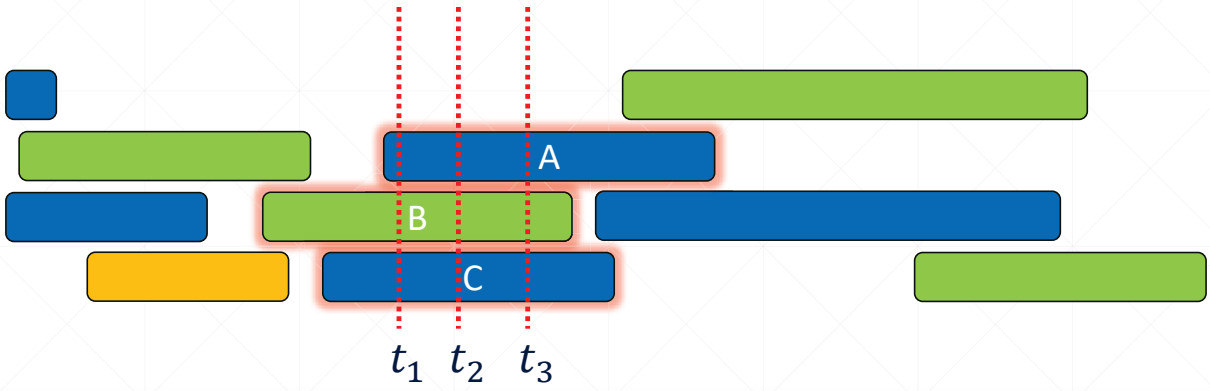
Questions

a. How to implement this algorithm efficiently?

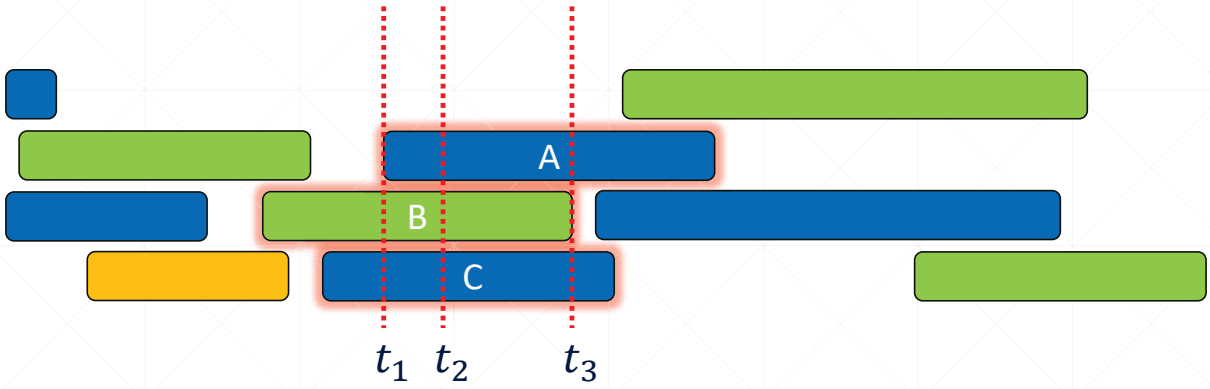
b. Why is this algorithm correct?

c. How to assign jobs to machines?

d. What is the running time of the algorithm?



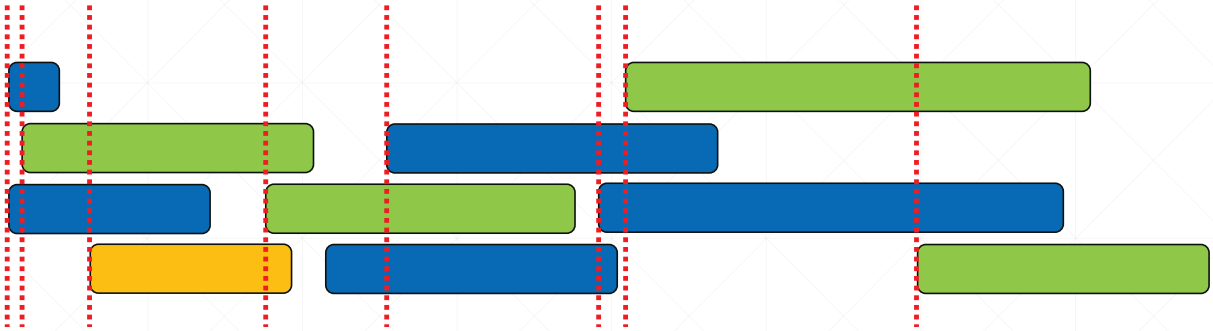
Observation: $m_{t_1} = m_{t_2} = m_{t_3}$



Observation: $m_{t_1} = m_{t_2} = m_{t_3}$

m_t changes only when t crosses $s(j)$ or $f(j)$ for some j.

Thus, it is sufficient to compute m_t only for $t \in \{s(j): j \in J\}$.



m_t changes only when t crosses $s(j)$ or $f(j)$ for some j.

Thus, it is sufficient to compute m_t only for $t \in \{s(j): j \in J\}$.

Sweep Line Algorithm*

Events = time points when a job starts or finishes

$$\mathcal{E} = \{s(j): j \in J\} \cup \{f(j): j \in J\}$$

$m_0 = 0$
Sort all events in increasing order.
For all events i in $\mathcal{E}$:

- If a job starts at event point $\#i$: $m_i = m_{i-1} + 1$.
- If a job finishes at event point $\#i$: $m_i = m_{i-1} - 1$.

Return $\max_i m_i$

Sweep Line Algorithm*

Events = time points when a job starts or finishes

$$\mathcal{E} = \{s(j): j \in J\} \cup \{f(j): j \in J\}$$

$m_0 = 0$

Sort all events in increasing order.

For all events i in $\mathcal{E}$:

- If a job starts at event point $\#i$: $m_i = m_{i-1} + 1$.
- If a job finishes at event point $\#i$: $m_i = m_{i-1} - 1$.

Return $\max_i m_i$

*We need to take a special care of jobs that start or finish at the same time.

Sweep Line Algorithm*

Events = time points when a job starts or finishes

$$\mathcal{E} = \{s(j): j \in J\} \cup \{f(j): j \in J\}$$

- $m_0 = 0$
- Sort all events in increasing order.
- For event i:
  - If a job starts at event point $\#i$: $m_i = m_{i-1} + 1$.
  - If a job finishes at event point $\#i$: $m_i = m_{i-1} - 1$.

Return $\max_i m_i$

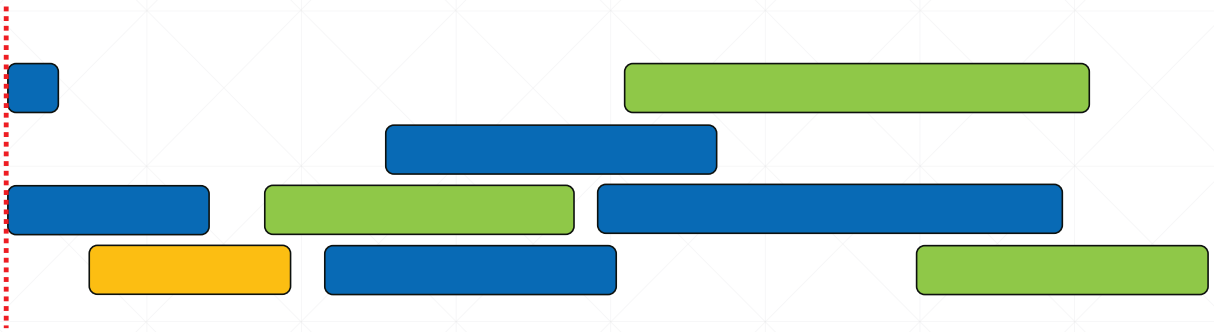
*We need to take a special care of jobs that start or finish at the same time.

Assigning Jobs to Machines

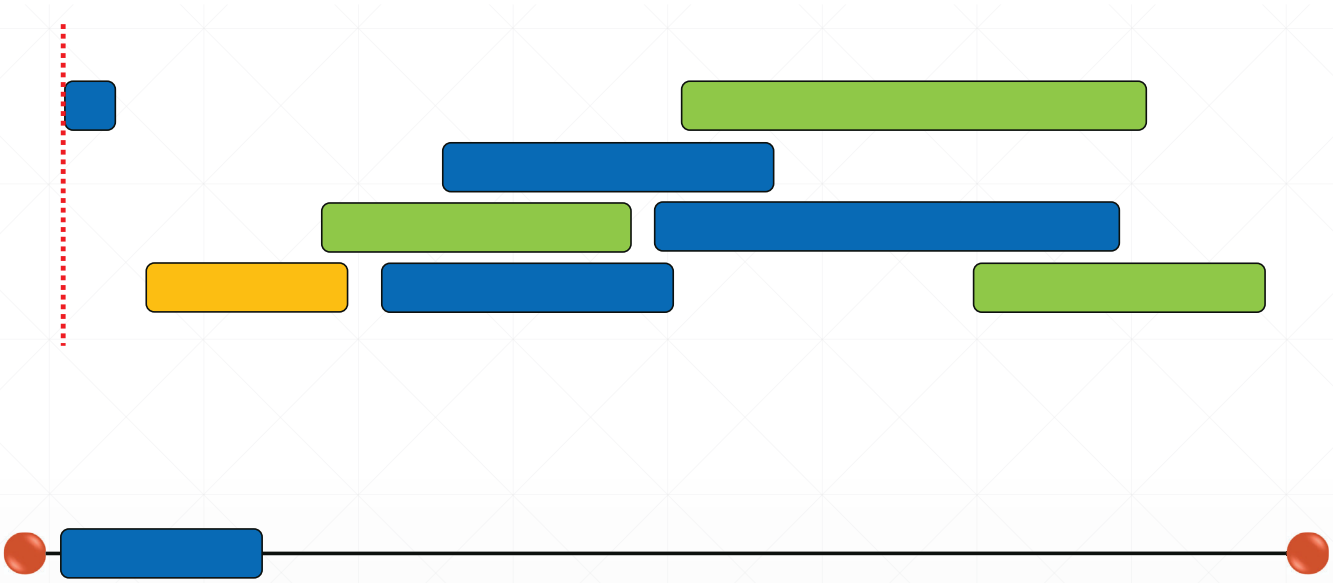
Sort all jobs in increasing order of $s(j)$.

for all j in J:

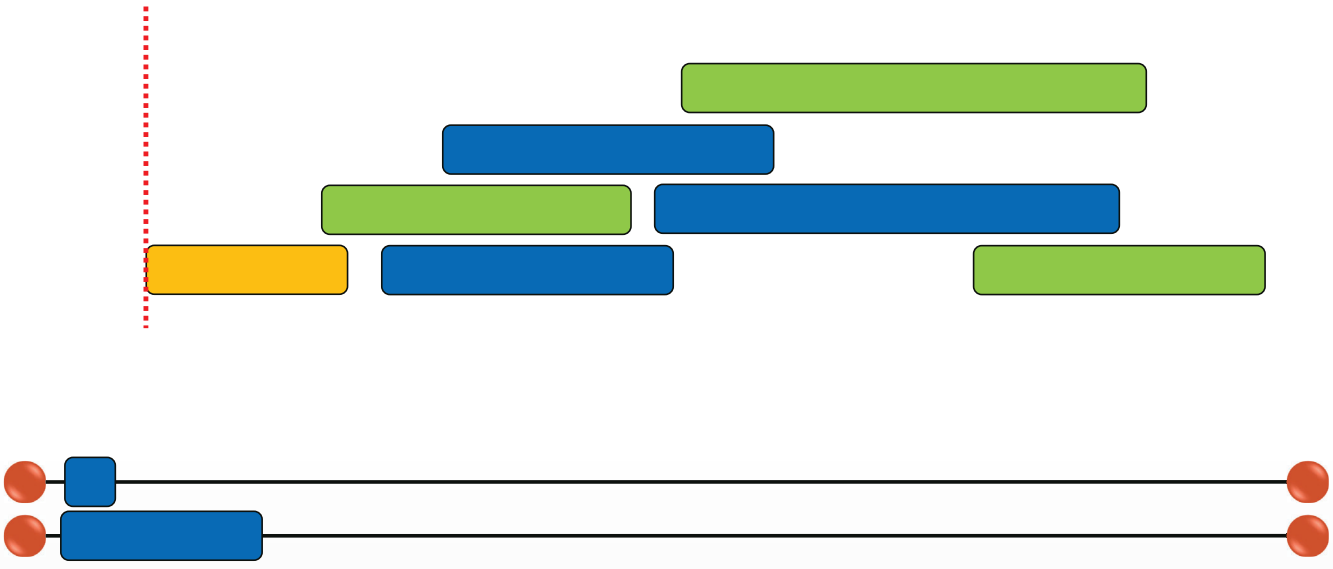
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



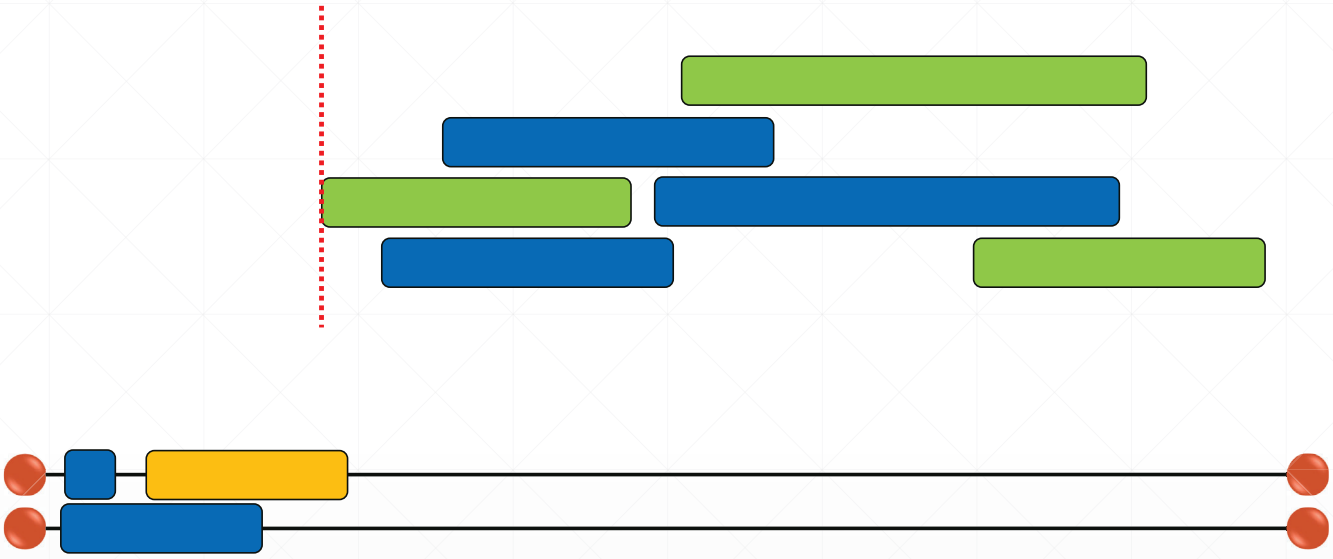
for all $j \in J$:
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



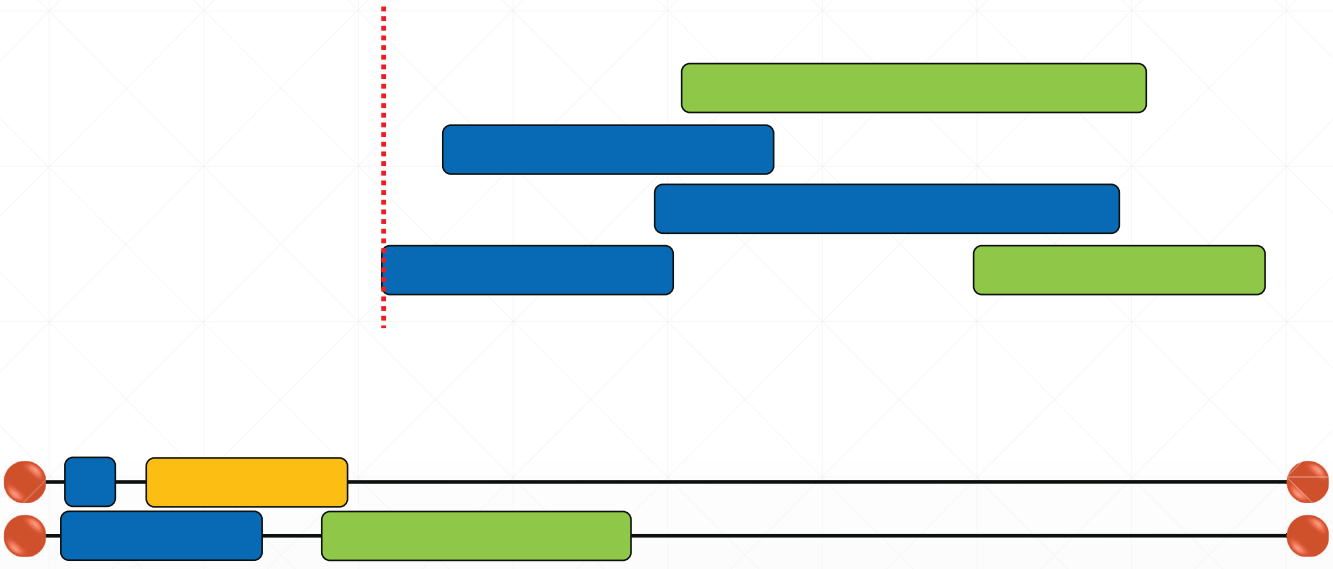
for all $j \in J$:
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



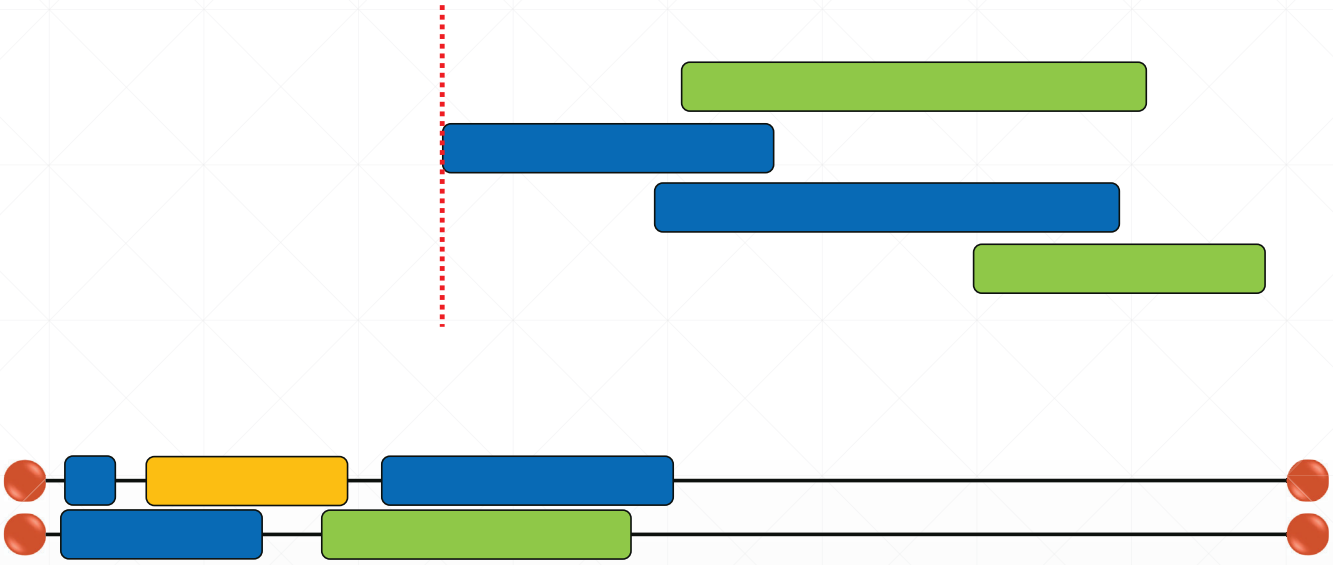
for all $j \in J$:
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



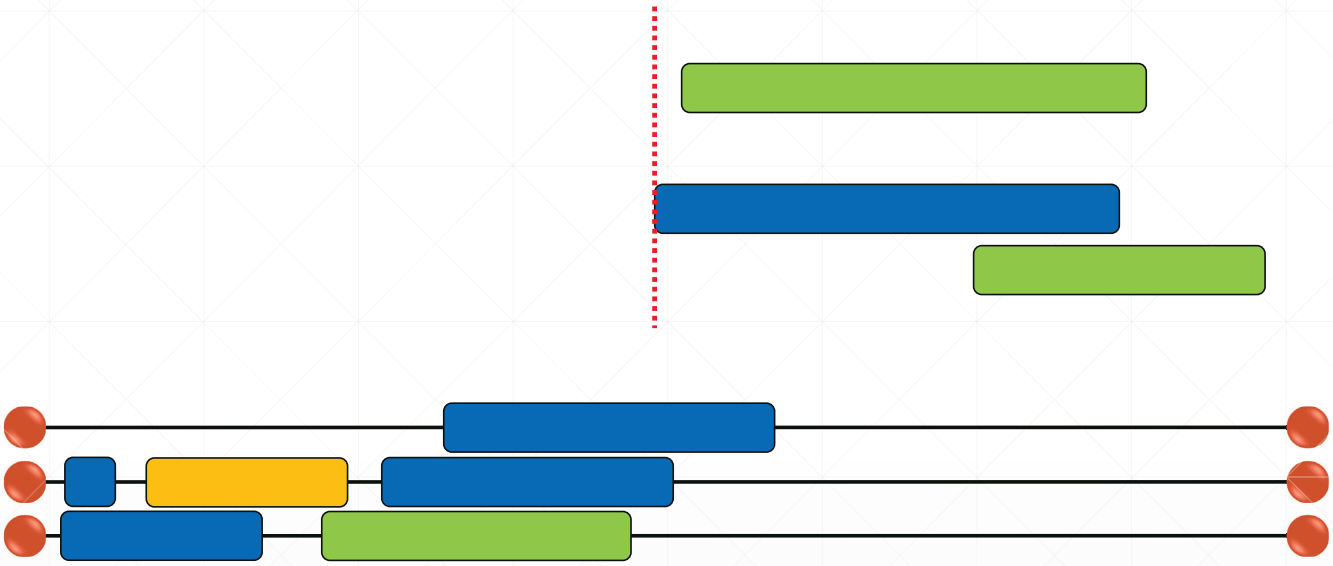
for all $j \in J$:
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



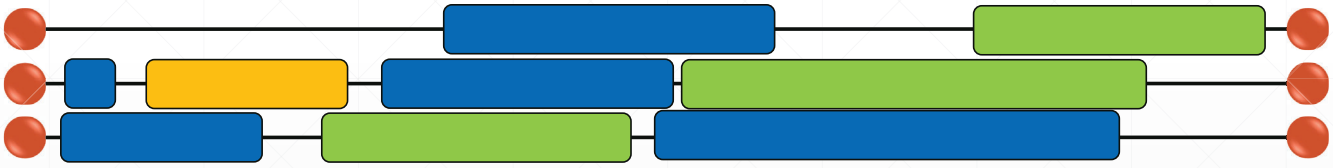
for all $j \in J$:
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



for all $j \in J$:
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



for all $j \in J$:
if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.



for all $j \in J$:
    if there is an available machine at time $s(j)$ assign j to that machine; otherwise, add a new machine.

Sweep Line Algorithm for Job Assignment

Sort set of all jobs J in increasing order of $s(j)$.
Priority queue for currently available machines Q ordered by time when machine i becomes available. $Q = \{1\}$.

for all $j \in J$:
  if ($Q.top.available \leq s(j)$): *//there is an available machine*
    Remove it from the queue: $i = Q.get$
  else
    Create a new machine i.

Assign job j to machine i.
Insert i in Q. Set time when i becomes available to $f(j)$.

Correctness

- The algorithm always finds a feasible schedule. That is, no two jobs are scheduled on the same machine at the same time.
- At time $s(j)$, exactly $m_{s(j)} - 1$ are busy, before we schedule job j. After we schedule job j, we use exactly $m_{s(j)}$ machines at time $s(j)$.

Running Time

- Sorting: $O(n \log n)$.
- We add and remove machines to Q at most n times
  - at times $s(j)$, where $j \in J$.
- The total cost of all priority queue (binary heap) operations is $O(n \log m)$.
- Also, at every iteration we perform $O(1)$ operations.
- We can create at most n machines, hence $m \leq n$.
- Total running time: $O(n \log n)$.

Conclusion

- Learned about
  - Machine Minimization (under Interval Scheduling assumptions).