CSE 331: Algorithms & Complexity

"Greedy Algorithms"

Prof. Charlie Anne Carlson (She/Her)

Lecture 15

Wednesday October 1st, 2025

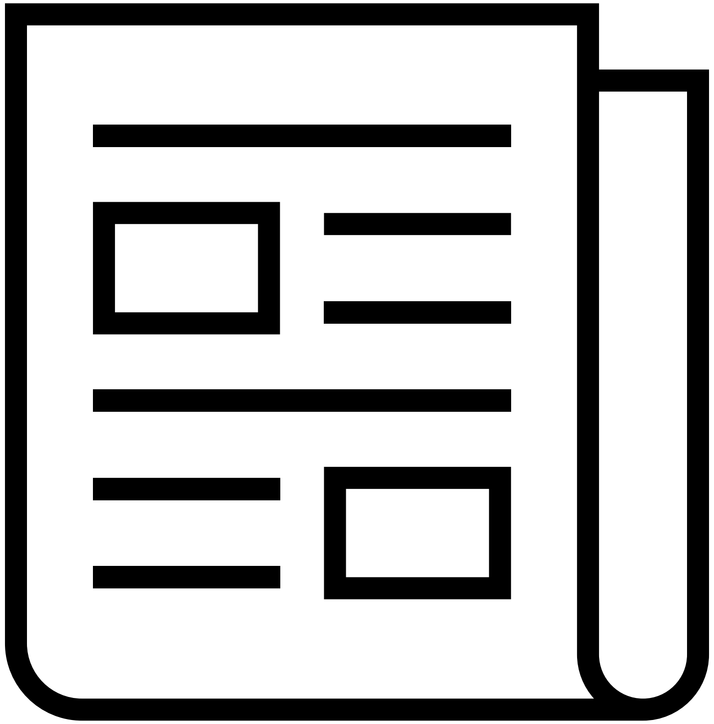
University at Buffalo

Schedule

1. Course Updates
2. Strong Connectivity
3. Greedy Algorithms
4. Change Making
5. Interval Scheduling

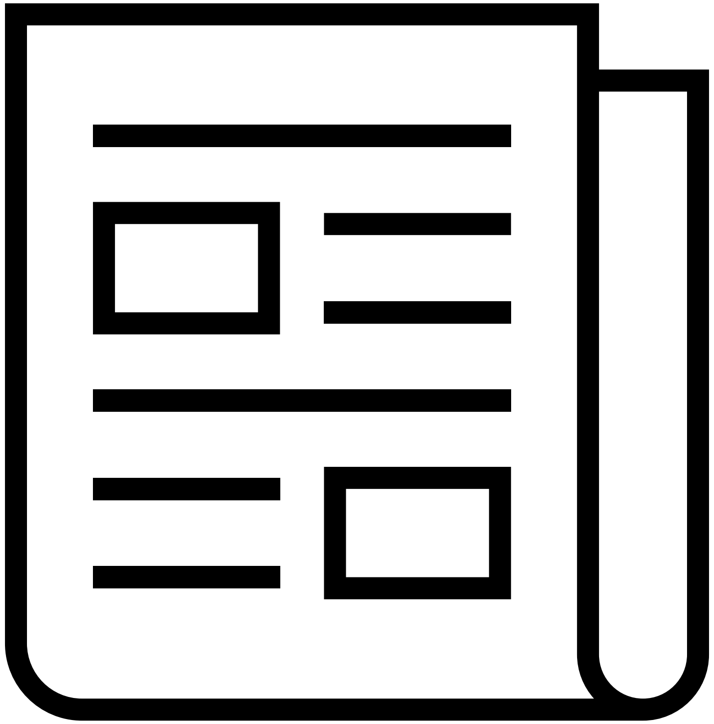
Course Updates

- HW 2 Grading Out
- HW 3 Solutions Out
- HW 4 Out Soon
    - Not Due Next Week!
- Group Project
    - First Problems Oct 31st
- Sample Midterms Out
- Midterms Oct 6 and Oct 8



Midterms

- Advice:
  - Start Studying
  - Go to Recitations this Week
  - Review Book Chapters
  - Review Solutions to HW/Quiz
  - Try Sample Midterm
  - Make Good Use of Time



Strong Connectivity Problem

Input: Directed graph $G = (V, E)$
Output: True if strongly connected and False otherwise.

Proof Idea:

- Pick a vertex s in V
- Use BFS to find all vertices I can reach from s.
- Use ______________ to find all vertices that can reach s.
- If both sets are equal return true and otherwise return false.

Strong Connectivity Algorithm

- **Definition**: We say that a directed graph is strongly connected if for any two vertices in the graph, there exists a directed path from one to the other.
- **Observation**: If u and v are mutually reachable, and v and w are mutually reachable, then u and w are mutually reachable.
- **Strong Connectivity Problem:**
  - **`Input`**`:` `Directed graph` $G = (V, E)$
  - **`Output`**`:` `True if strongly connected and False otherwise.`

Strong Connectivity Algorithm

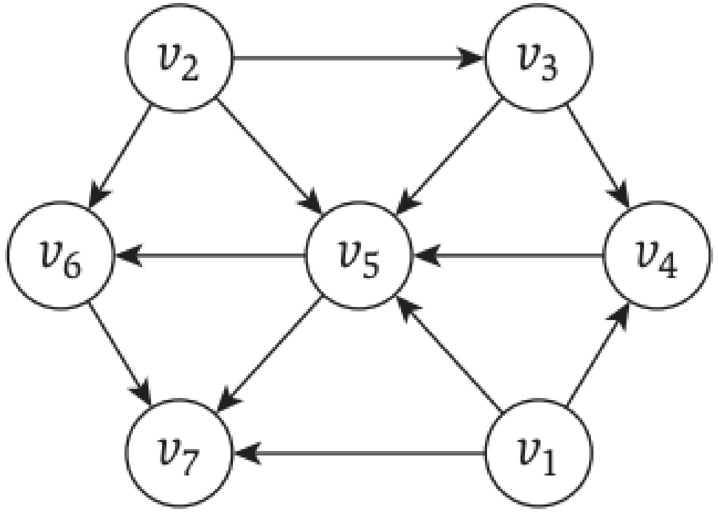
Input: Directed graph $G = (V, E)$
Output: True if strongly connected and False otherwise.

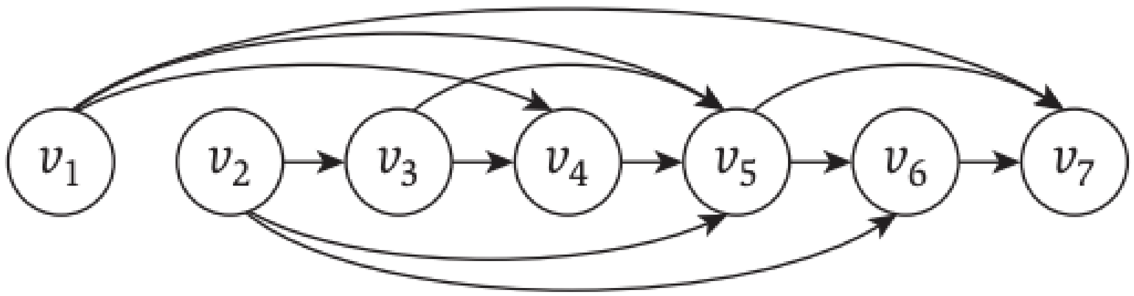
Proof Idea:

- Pick a vertex s in V
- Use BFS to find all vertices I can reach from s.
- Use BFS on "Reversed Graph" to find all vertices that can reach s. For each edge (u,v) replace with edge (v,u)
- If both sets are equal return true and otherwise return false.

Directed Acyclic Graphs (DAGs)

- **Definition:** A directed graph is a DAG if it has no directed cycles.
- **Definition:** A topological ordering of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j), we have i < j.

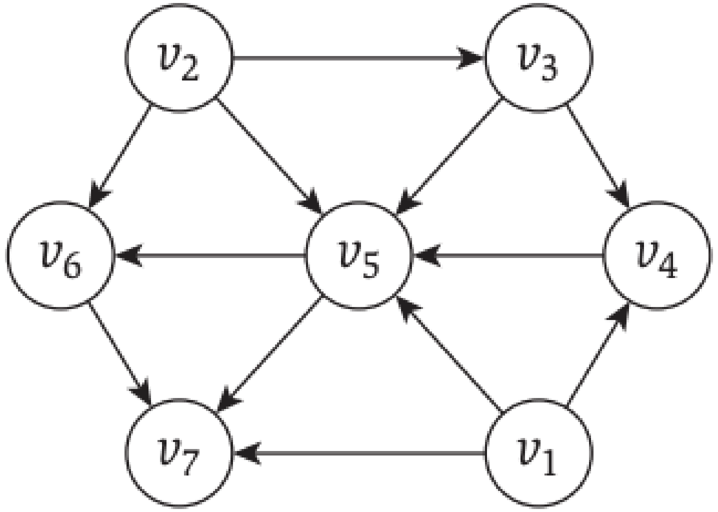


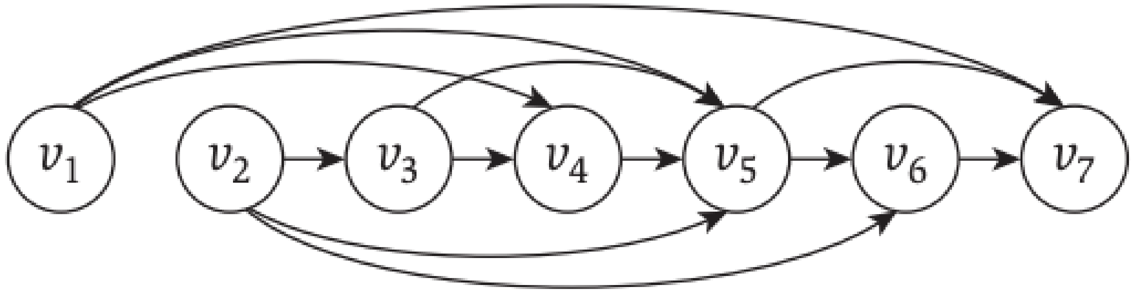


Directed Acyclic Graphs (DAGs)

Definition: A directed graph is a DAG if it has no directed cycles.

Definition: A topological ordering of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j), we have i < j.





All edges are going “forward”

Directed Acyclic Graphs (DAGs)

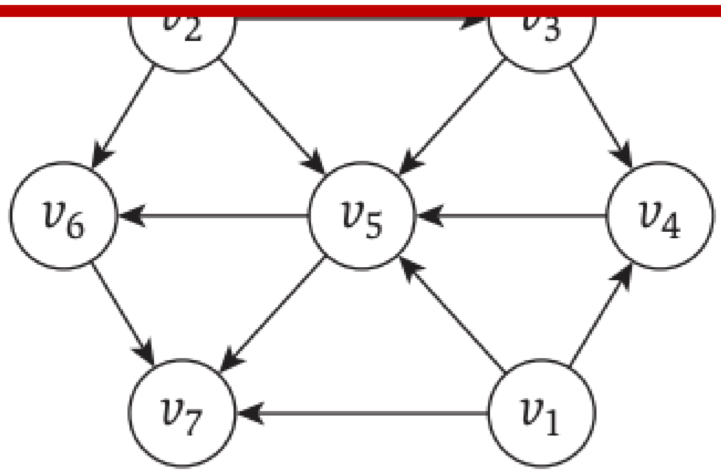
Definition: A directed graph is a DAG if it has no directed cycles.

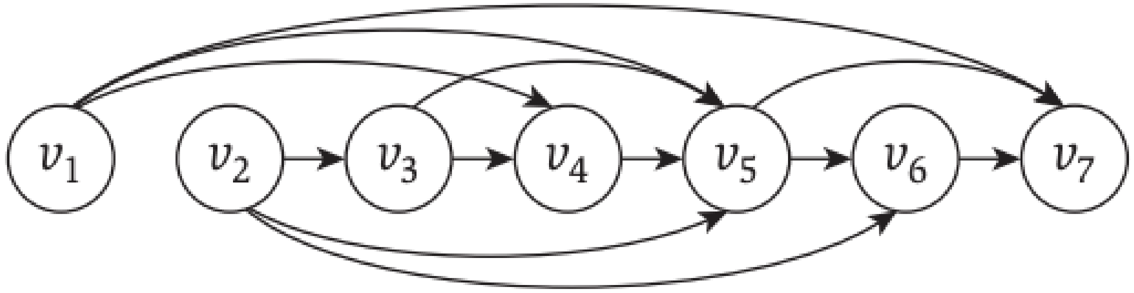
Definition: A topological ordering of a directed graph $G =$

(

e

Read KT Section 3.6 and Review Care Packaged on Topological Ordering





All edges are going “forward”

Midterm Check Point

What is next?

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
- Network Flows (maybe)
- Computation Complexity

“How do we design new algorithms?”

- **Greedy Algorithms**
- **Divide and Conquer**
- **Dynamic Programming**
- **Network Flows (maybe)**
- Computation Complexity

“How do we use reduce another problem?”

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
- **Network Flows (maybe)**
- **Computation Complexity**

"How do we know when to give up?"

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
- Network Flows (maybe)
- **Computation Complexity**

What are Greedy Algorithm?

What are Greedy Algorithm?

- Build solution one piece at a time.
- Only look at immediate information to make choices.
- Never go back on a decision.
- NOT ALWAYS THE BEST CHOICE!

Coin Change Problem

- **Problem**: Given U.S. currency denominations {1.00, 0.25, 0.10, 0.05, 0.01} find an algorithm to pay an amount to a customer using the fewest coins possible.

Coin Change Problem

- **Problem**: Given U.S. currency denominations {1.00, 0.25, 0.10, 0.05, 0.01} find an algorithm to pay an amount to a customer using the fewest coins possible.

- **Algorithm**: At each iteration, add a coin of the largest value that is less than the amount needed to be paid.

Q: Is this algorithm always optimal?

- **Problem**: Given U.S. currency denominations {1.00, 0.25, 0.10, 0.05, 0.01} find an algorithm to pay an amount to a customer using the fewest coins possible.

- **Algorithm**: At each iteration, add a coin of the largest value that is less than the amount needed to be paid.

Proof Ideas for Optimality

- **Proof Idea:**
  - Suppose there it wasn't optimal.
    - Then there exists a budget B such that the algorithm returns S and the answer is S' (S != S').

Proof Ideas for Optimality

- **Proof Ideas:**
  - Suppose there it wasn’t optimal.
    - Then there exists a budget B such that the algorithm returns a set S(B) and the answer is a different set S’(B).
  - If S’(B) has the largest coin that S(B) has, then you can remove it and you get a smaller bad budget B’ = B - <large coin value>

Proof Ideas for Optimality

- **Proof Ideas:**
  - Suppose there it wasn’t optimal.
    - Then there exists a budget B such that the algorithm returns a set S(B) and the answer is a different set S’(B).
  - If S’(B) has the largest coin that S(B) has, then you can remove it and you get a smaller bad budget B’ = B - <large coin value> ← Use Induction on That!

Proof Ideas for Optimality

- **Proof Ideas:**
  - If S’(B) has the largest coin that S(B) has, then you can remove it and you get a smaller bad budget B’ = B - <large coin value>
  - We now show that S’(B) has to have the largest coin that S(B) has.
    - That is, the greedy choice was good!

Proof Ideas for Optimality

- **Proof Ideas:**
  - We now show that S'(B) has to have the largest coin that S(B) has.
  - If S'(B) doesn't have the largest coin that can fit in the budget, then it must be replaced with smaller coins.
    - We can check optimal for restricted settings of coins. ← Sounds like a few base cases

Proof Ideas for Optimality

- **Proof Ideas:**
  - We can show that in an optimal solution we have:
    - At most 4 pennies
    - At most 1 nickel
    - At most 2 nickels + dimes
    - At most 3 quarters
  - We show these by contradicting the optimality!

← Sounds like a few base cases

Proof Ideas for Optimality

- **Proof Ideas:**
  - We can show that in an optimal solution we have:
    - At most 4 pennies
    - At most 1 nickel
    - At most 2 nickels + dimes
    - At most 3 quarters
  - We show these by contradicting the optimality!

← Sounds like a few base cases

Big Ideas

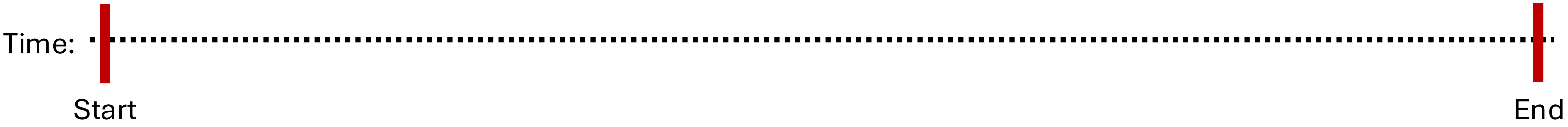
- Sometimes Greedy Works but sometimes it doesn’t…
  - If you use a different set of coins, you may not be able to use the cashier algorithm.
  - Consider {1,10,21,34,70,100,350,1225} and the budget 140.

Big Ideas

- Sometimes Greedy Works but sometimes it doesn't…
  - If you use a different set of coins, you may not be able to use the cashier algorithm.
  - Consider {1,10,21,34,70,100,350,1225} and the budget 140.
    - Algorithm Output: 100, 34, 1x6
    - Answer: 70 x 2

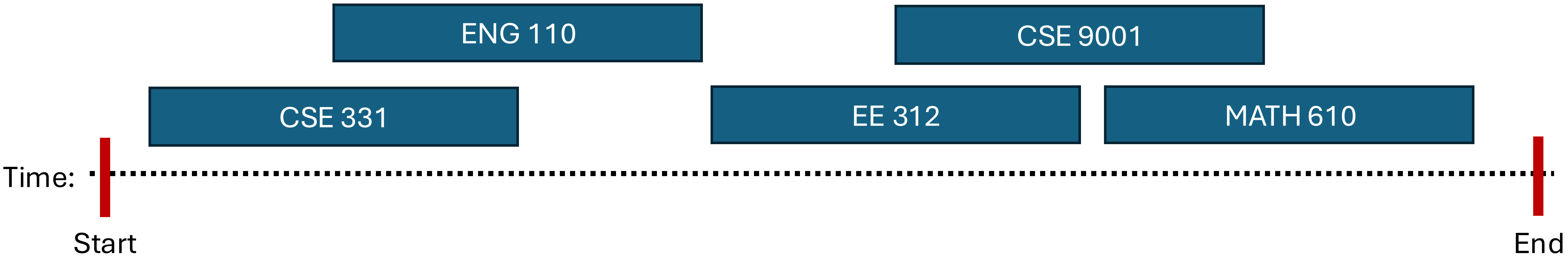
Interval Scheduling

- Consider an interval of time (e.g. Wednesday).



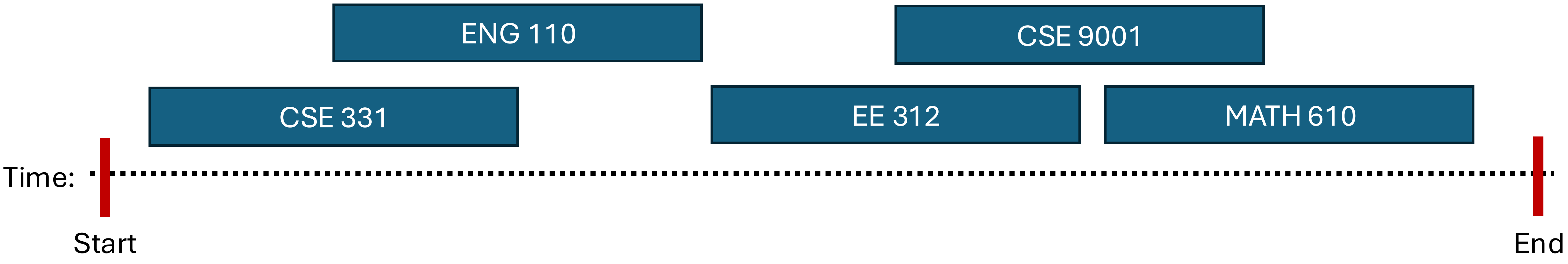
Interval Scheduling

- Consider an interval of time (e.g. Wednesday)
- Consider tasks that need to be completed during specific times (e.g. classes)



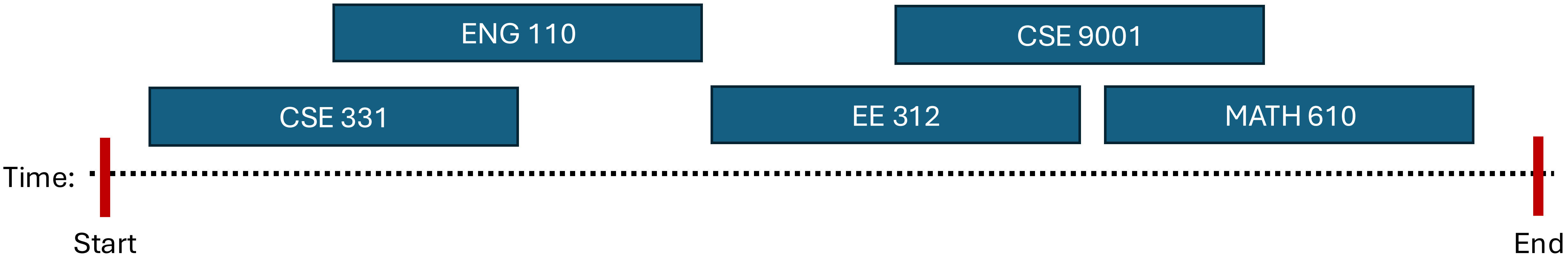
Interval Scheduling

- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.



Interval Scheduling

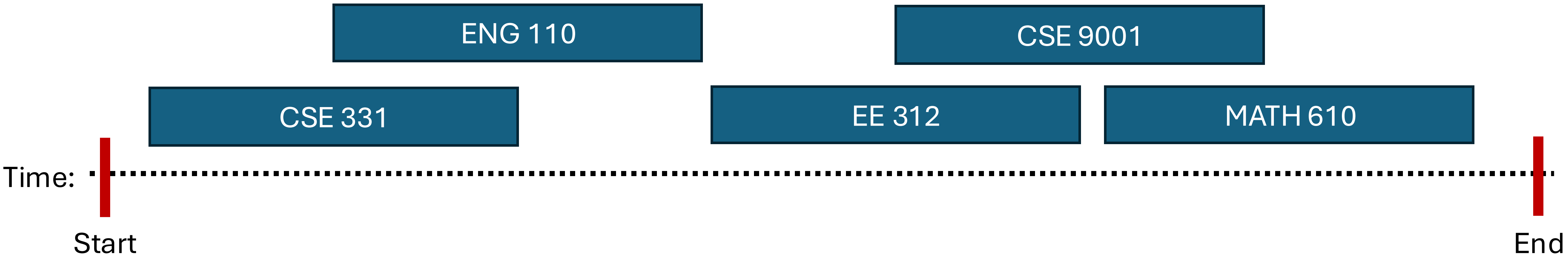
- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.



Interval Scheduling

- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.

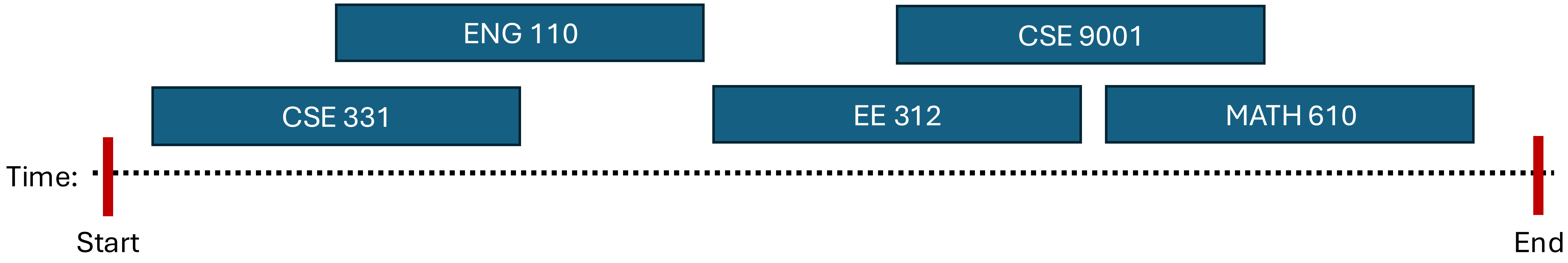
Q: Do we ever pick CSE 9001?



Interval Scheduling

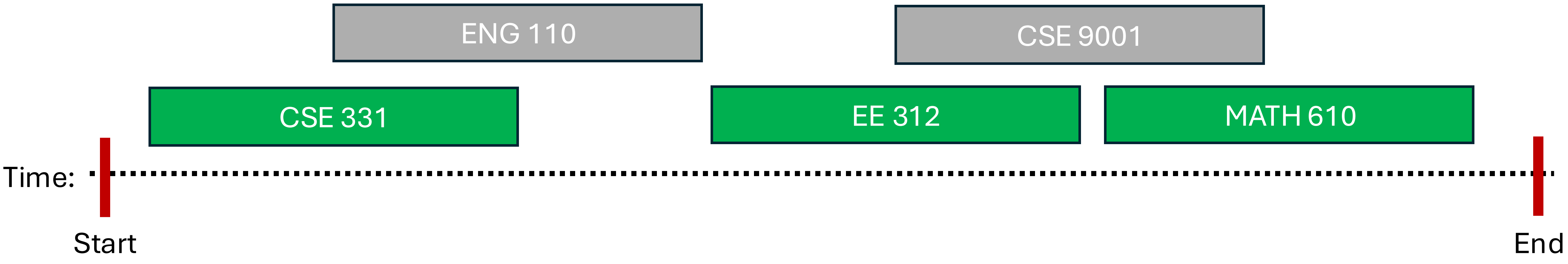
- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.

A: No, because it blocks two classes!



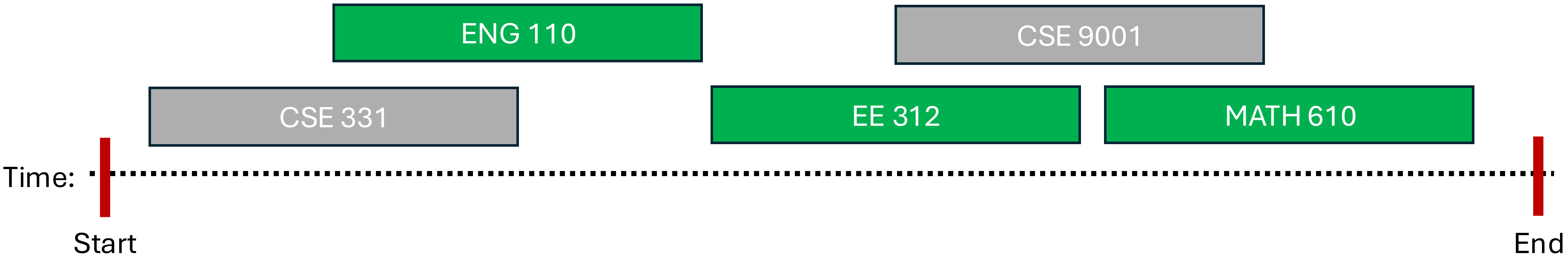
Optimal Solution #1

- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.



Optimal Solution #2

- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.



Interval Scheduling Problem (Support Page)

Interval Scheduling via examples

In which we derive an algorithm that solves the Interval Scheduling problem via a sequence of examples.

The problem

In these notes we will solve the following problem:

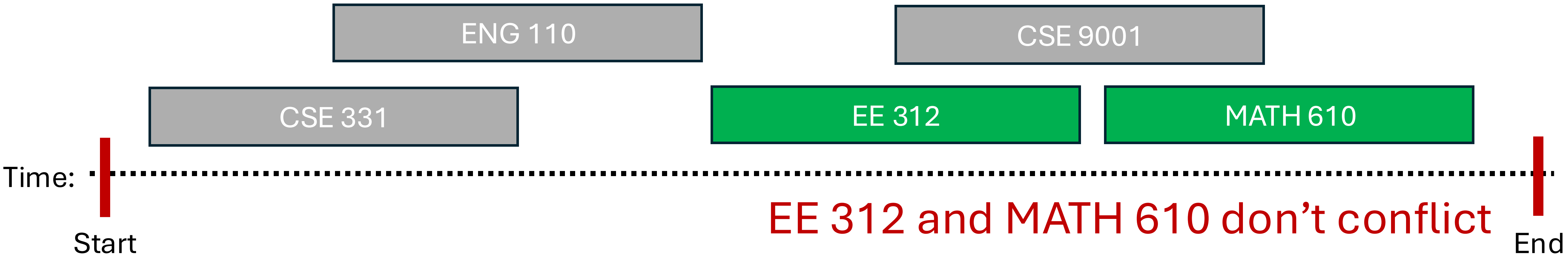
Interval Scheduling Problem

`Input:` An input of n intervals $[s(i), f(i))$, or in other words, $\{s(i), \ldots, f(i) - 1\}$ for $1 \leq i \leq n$ where i represents the intervals, $s(i)$ represents the start time, and $f(i)$ represents the finish time.

`Output:` A schedule S of n intervals where no two intervals in S conflict, and the total number of intervals in S is maximized.

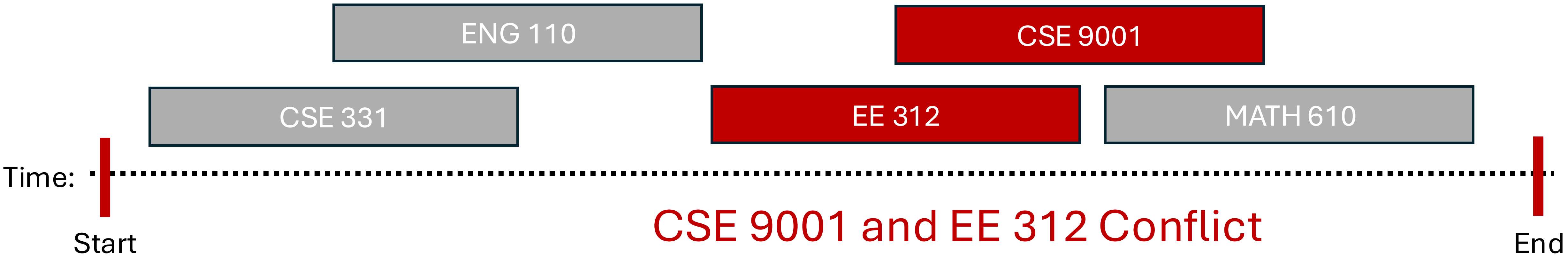
Interval Scheduling Problem

- **Input**: A set of n intervals with start and finish times.
  - For $1 \le \mathrm{i} \le \mathrm{n}$, $[s(\mathrm{i}), \mathrm{f}(\mathrm{i}))$ where $s(\mathrm{i})$ and $\mathrm{f}(\mathrm{i})$ are start and finish times of task i respectively.
- **Output**: A schedule (subset of intervals) S such that no two intervals in S conflict and the total number of intervals is maximized.



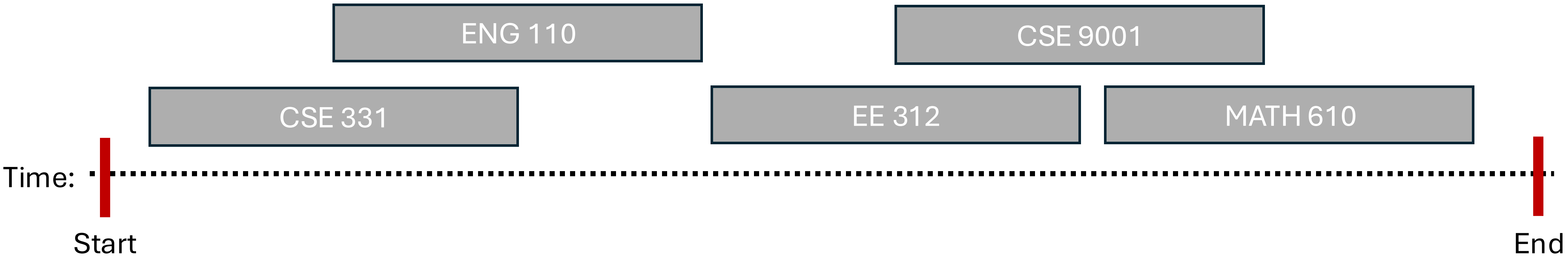
Interval Scheduling Problem

- **Input**: A set of n intervals with start and finish times.
  - For $1 \leq \mathrm{i} \leq \mathrm{n}$, $[s(\mathrm{i}), \mathrm{f}(\mathrm{i}))$ where $s(\mathrm{i})$ and $\mathrm{f}(\mathrm{i})$ are start and finish times of task i respectively.
- **Output**: A schedule (subset of intervals) S such that no two intervals in S conflict and the total number of intervals is maximized.



Q: How should we try to solve this?

- **Input**: A set of n intervals R with start and finish times.
  - For $1 \leq \mathrm{i} \leq \mathrm{n}$, $[s(\mathrm{i}), \mathrm{f}(\mathrm{i}))$ where $s(\mathrm{i})$ and $\mathrm{f}(\mathrm{i})$ are start and finish times of task i respectively.
- **Output**: A schedule (subset of intervals) S such that no two intervals in S conflict and the total number of intervals is maximized.



Q: How should we try to solve this?

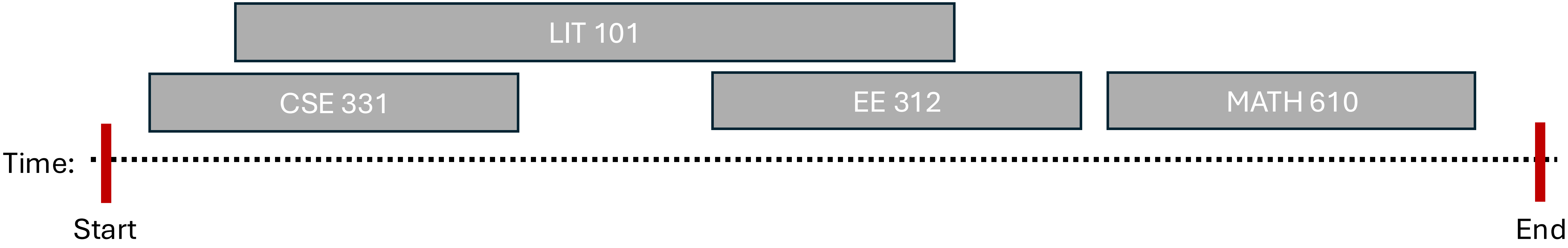
A: Let’s try to generate some examples and see what works and what doesn’t.

Build an Algorithm

- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R
    - Add i to S
    - Remove i from R

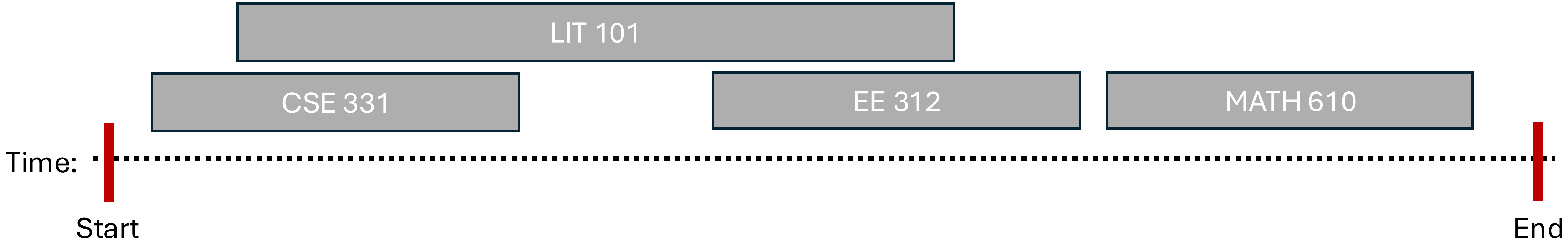
Build an Algorithm

- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R
    - Add i to S You might add conflicts!
    - Remove i from R



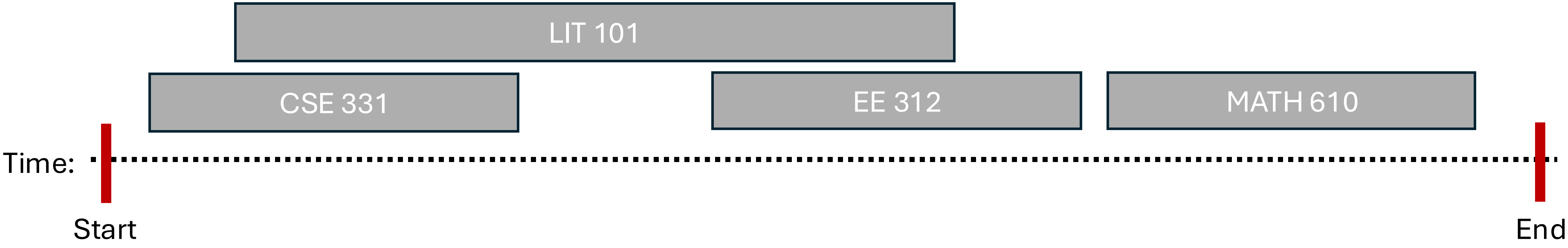
Build an Algorithm

- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R
    - Add i to S
    - Remove **all tasks that conflict with** i from R



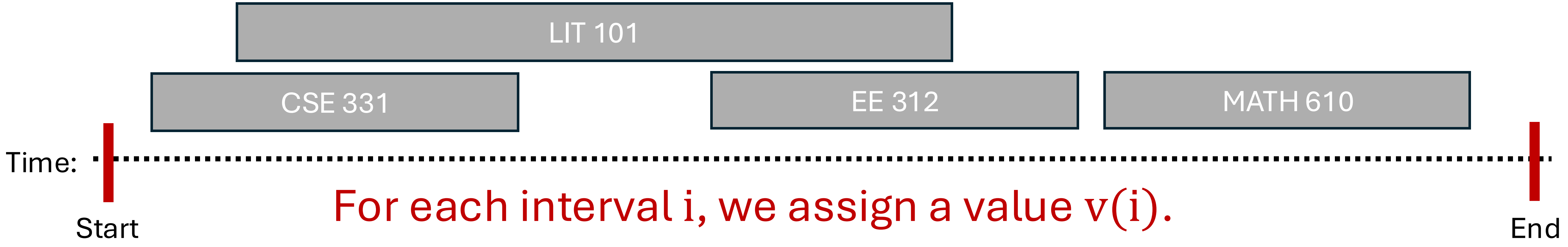
Build an Algorithm

- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R   Q: How do we do this?
    - Add i to S
    - Remove **all tasks that conflict with** i from R



Build a Greedy Algorithm

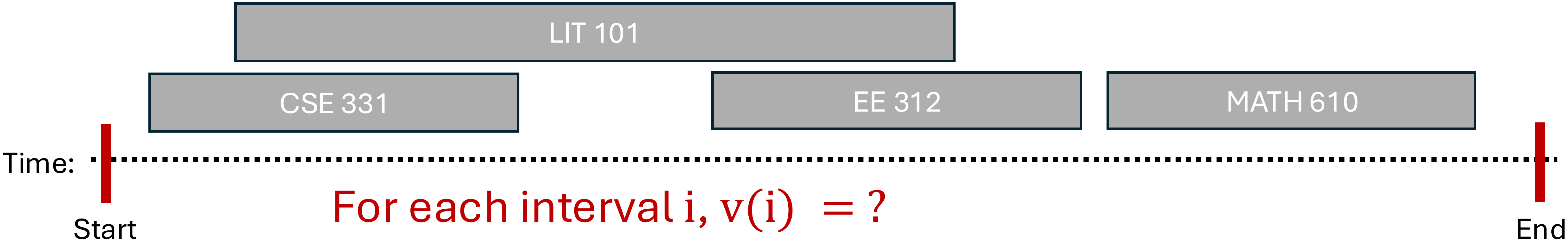
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove **all tasks that conflict with** i from R



For each interval i, we assign a value v(i).

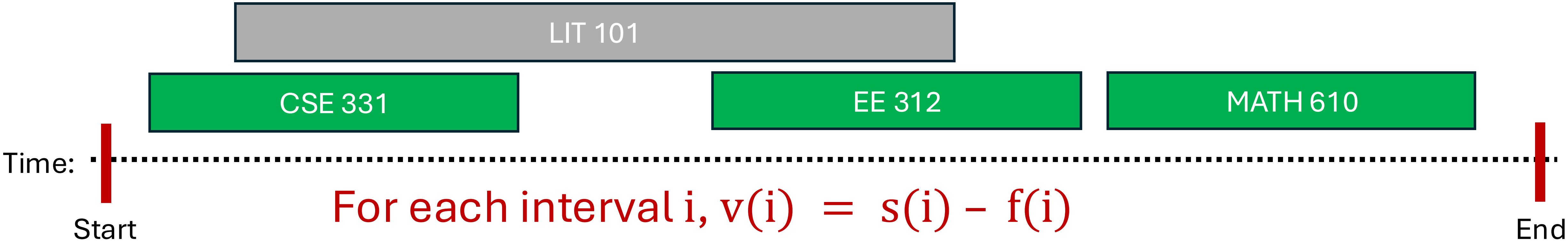
Q: What should we pick for $v(i)$?

- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



Attempt I: Interval Length (Okay)

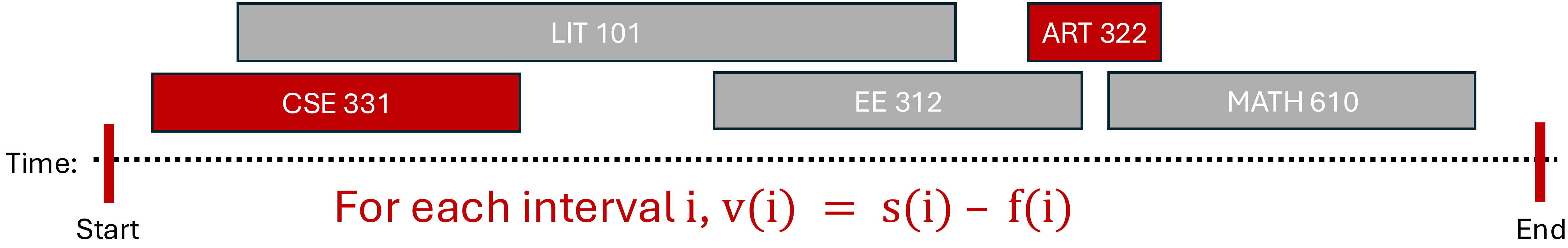
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



For each interval i, $v(i) = s(i) - f(i)$

Attempt I: Interval Length (Oh no!)

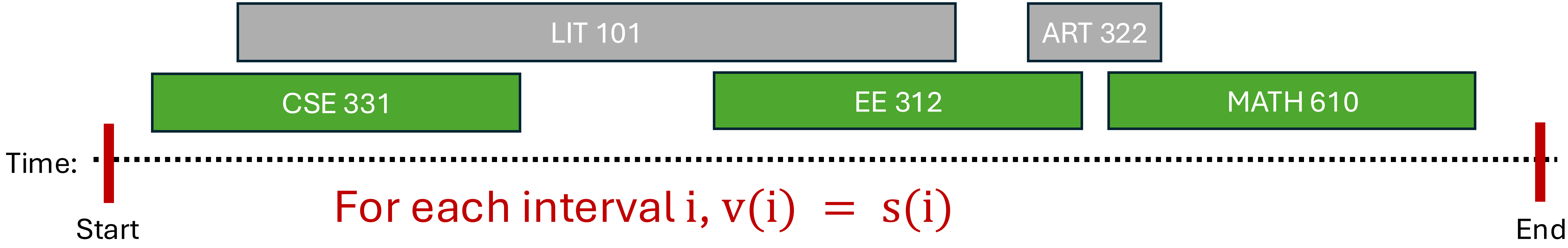
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



For each interval i, $v(i) = s(i) - f(i)$

Attempt II: Start Time (Okay)

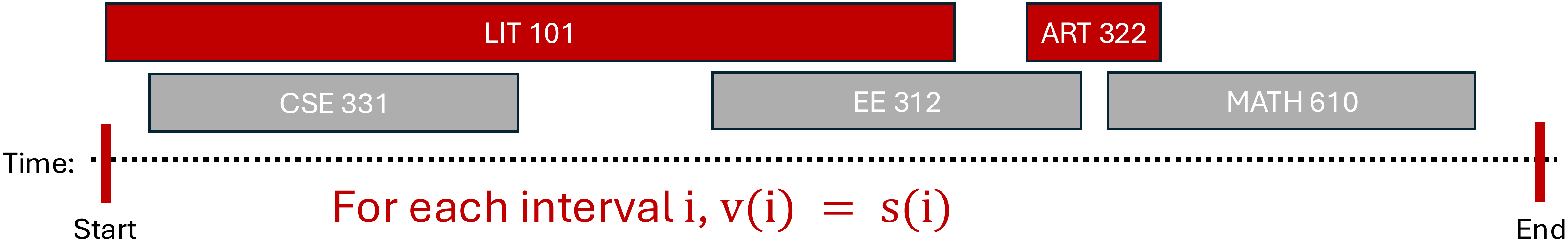
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



For each interval i, $v(i) = s(i)$

Attempt II: Start Time (Oh no!)

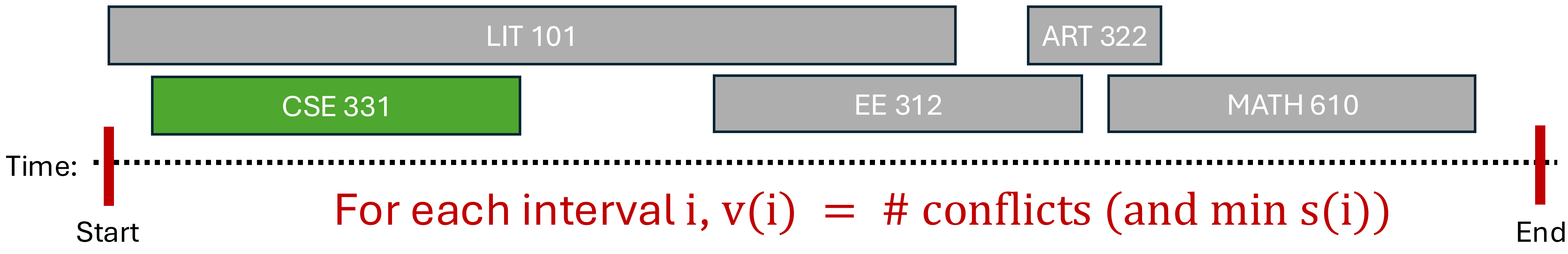
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



For each interval i, $v(i) = s(i)$

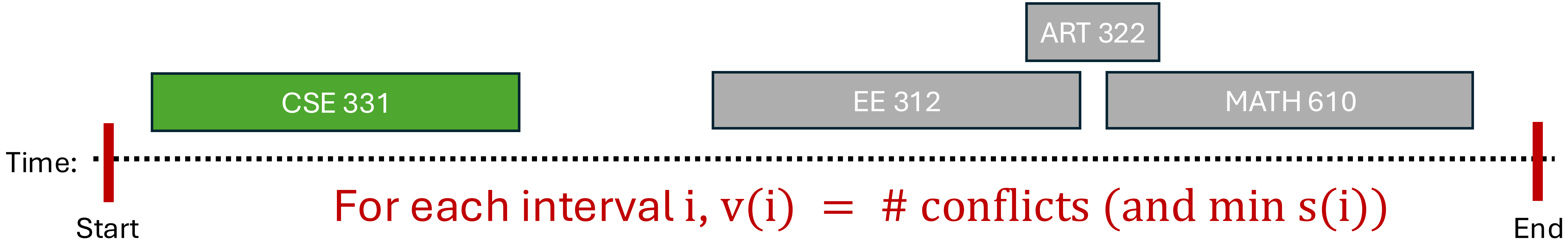
Attempt III: Min Conflicts (Okay)

- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



Attempt III: Min Conflicts (Okay)

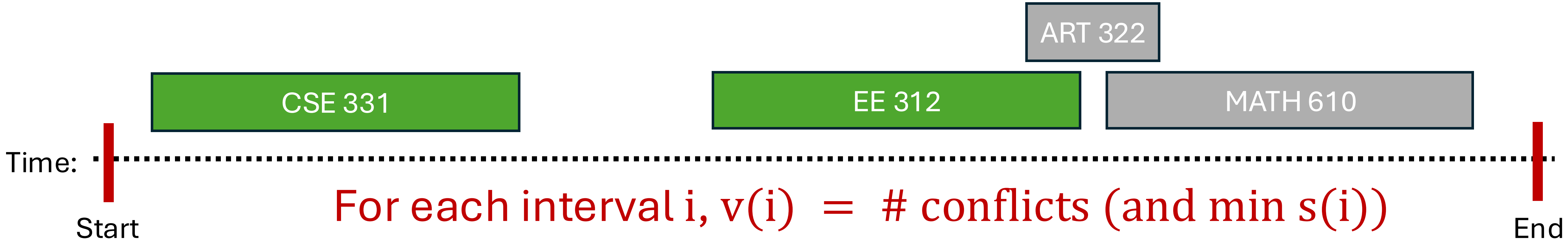
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Okay)

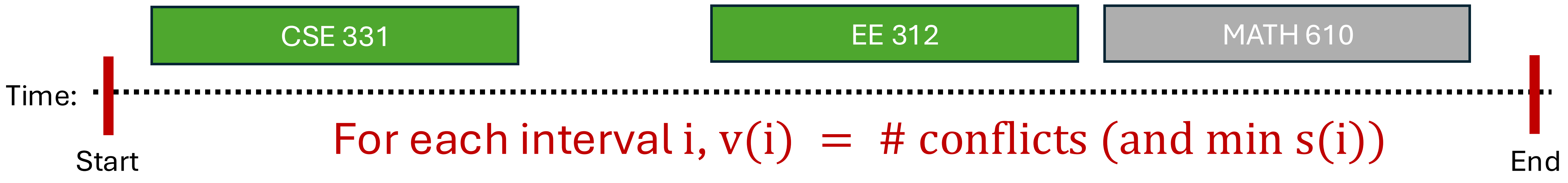
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



For each interval i, $v(i) = \#$ conflicts (and min $s(i)$)

Attempt III: Min Conflicts (Okay)

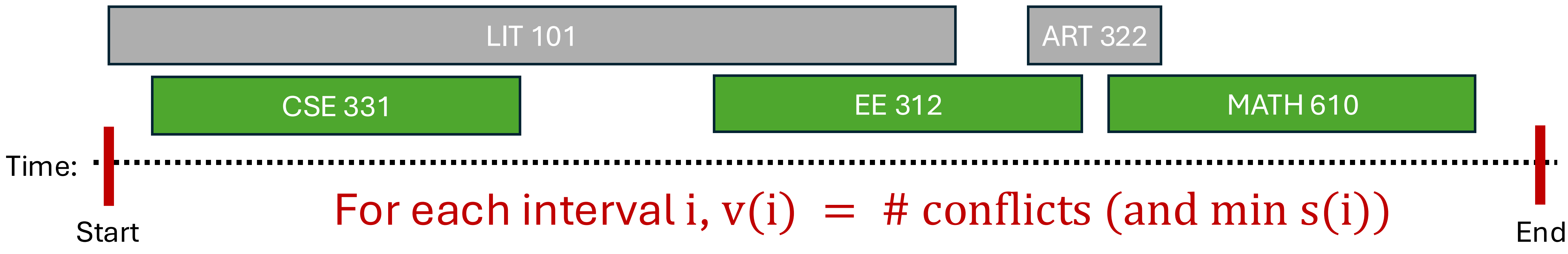
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



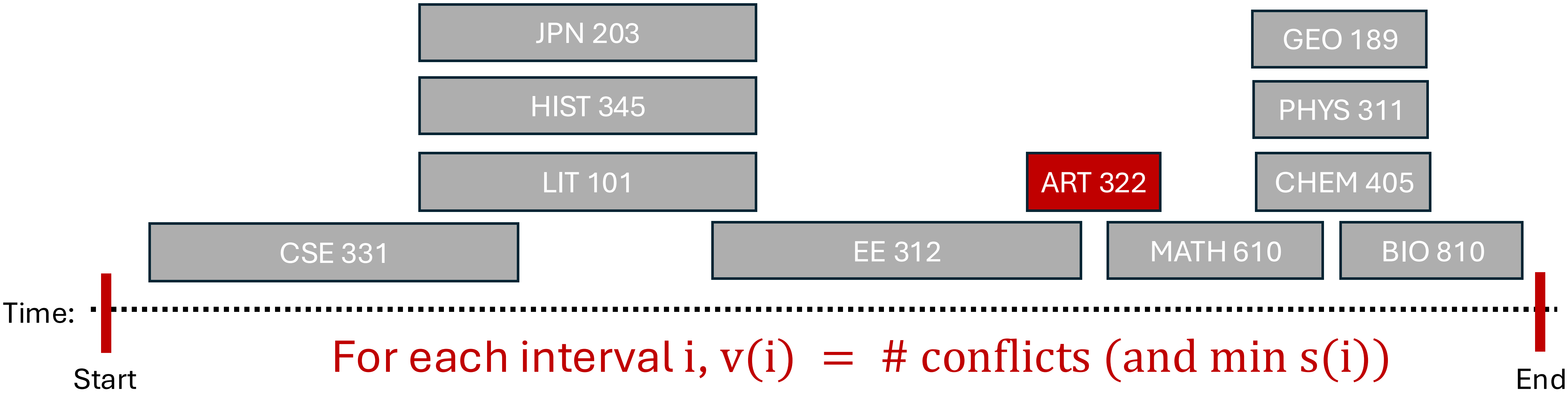
For each interval i, $v(i) = \#$ conflicts (and min $s(i)$)

Attempt III: Min Conflicts (Okay)

- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R

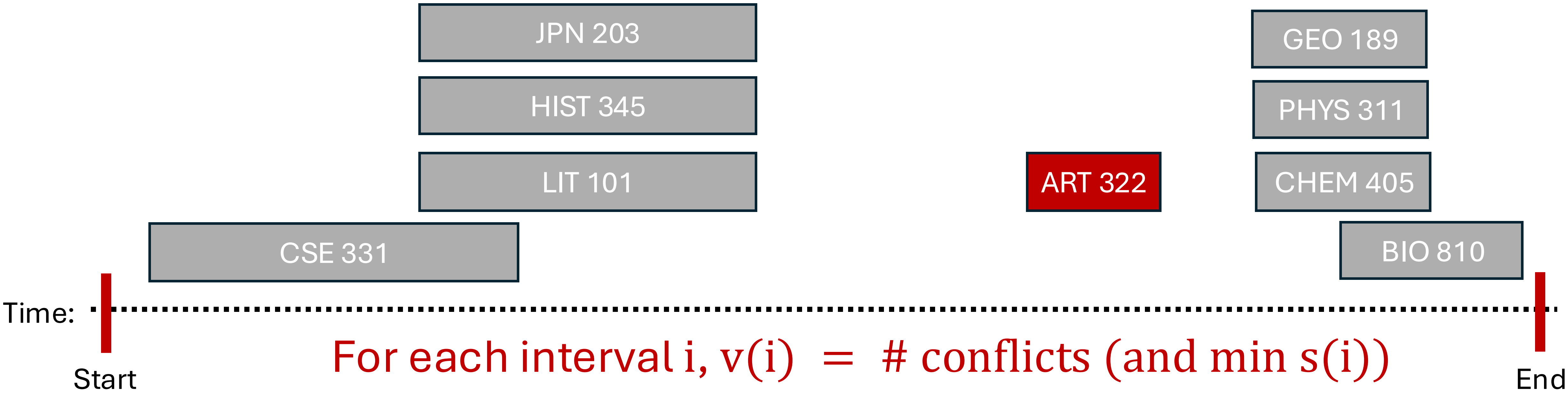


Attempt III: Min Conflicts (Oh no!)



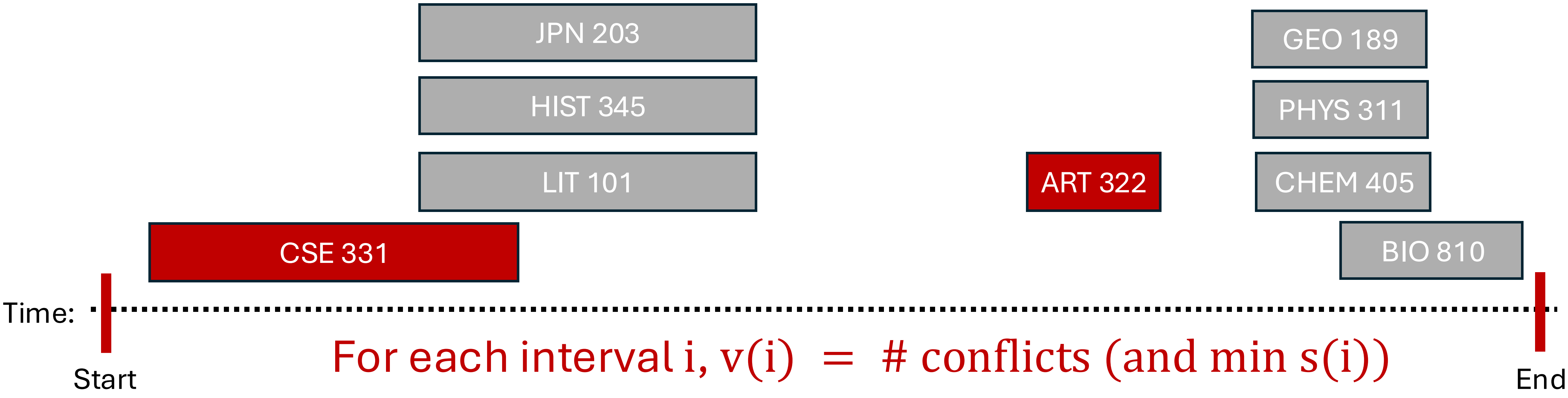
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



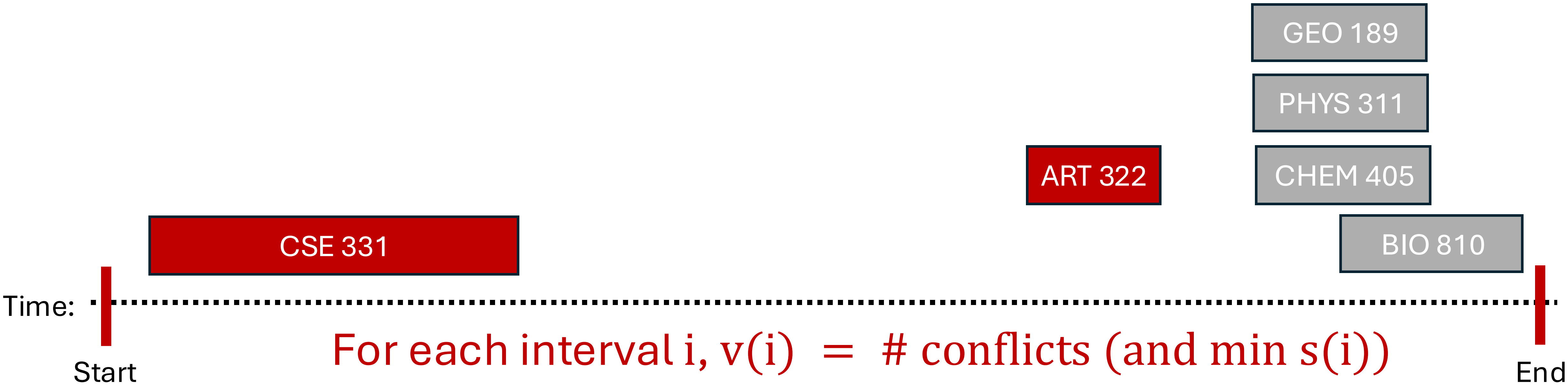
For each interval i, $v(i) = \#$ conflicts (and min $s(i)$)

Attempt III: Min Conflicts (Oh no!)



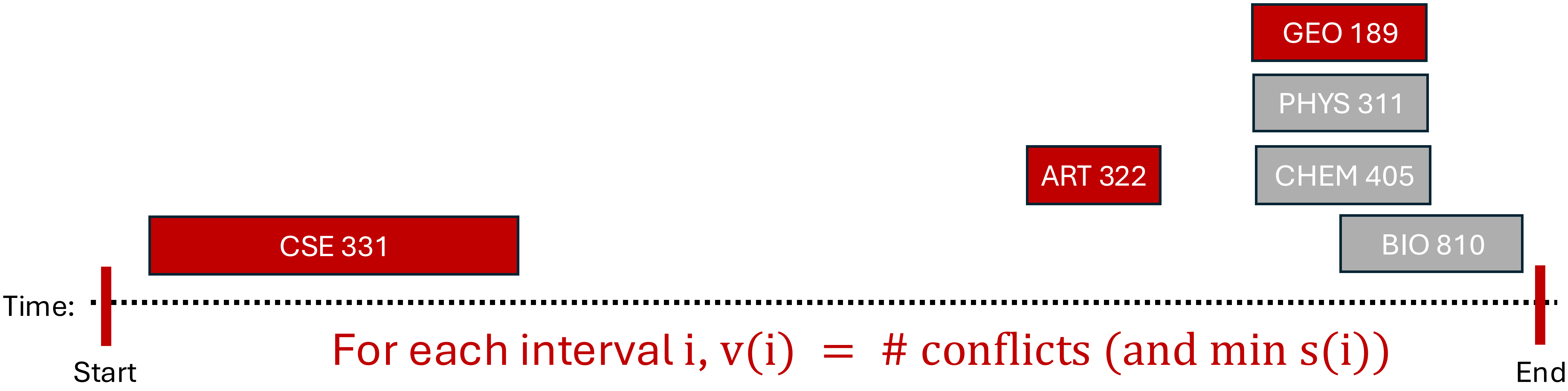
For each interval i, $v(i) = \#$ conflicts (and min $s(i)$)

Attempt III: Min Conflicts (Oh no!)



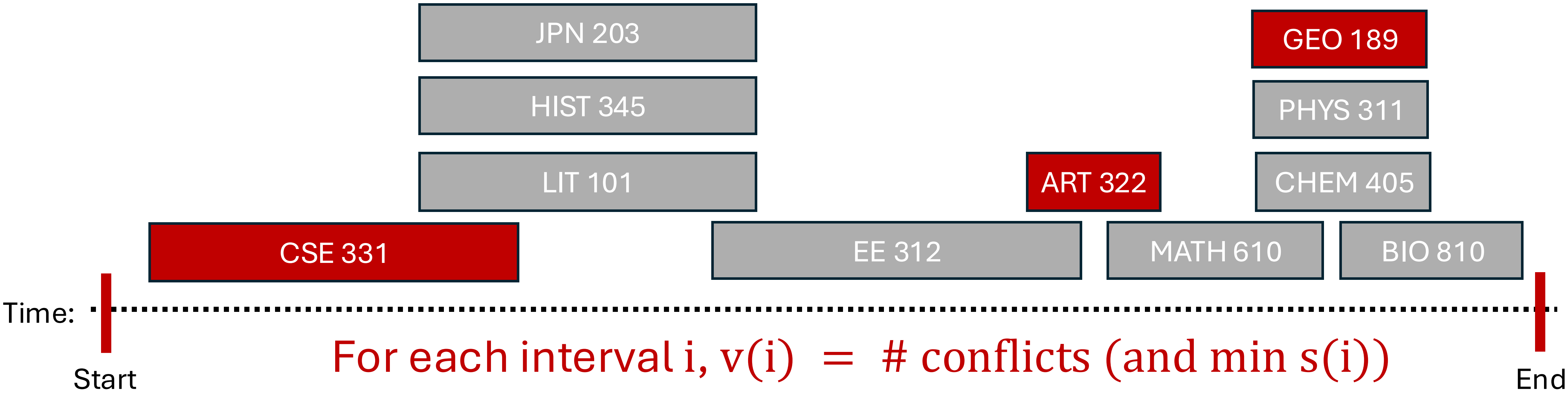
For each interval i, $v(i) = \#$ conflicts (and min $s(i)$)

Attempt III: Min Conflicts (Oh no!)



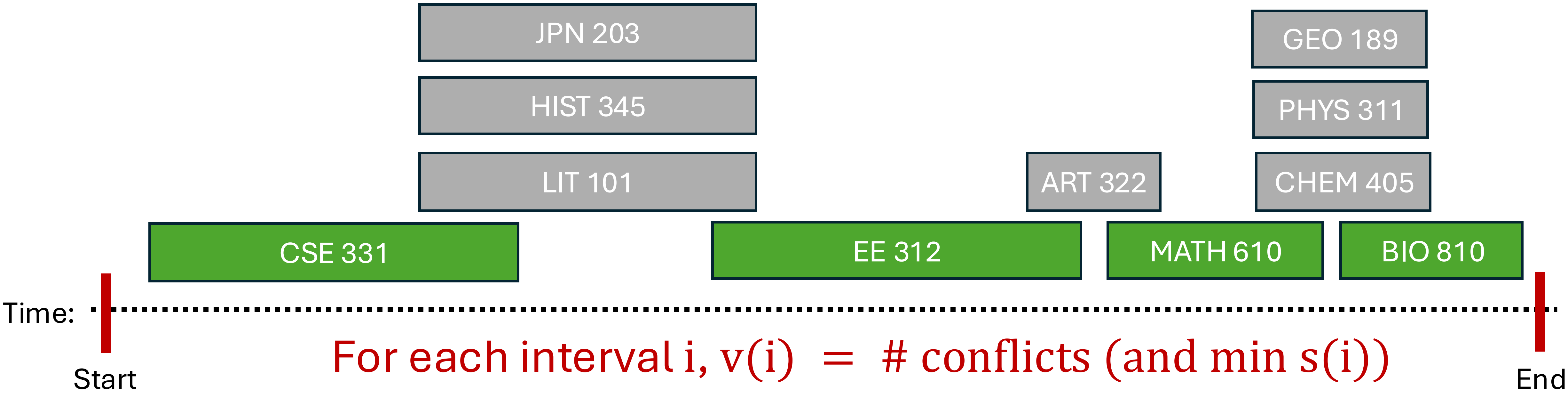
For each interval i, $v(i) = \#$ conflicts (and min $s(i)$)

Attempt III: Min Conflicts (Oh no!)



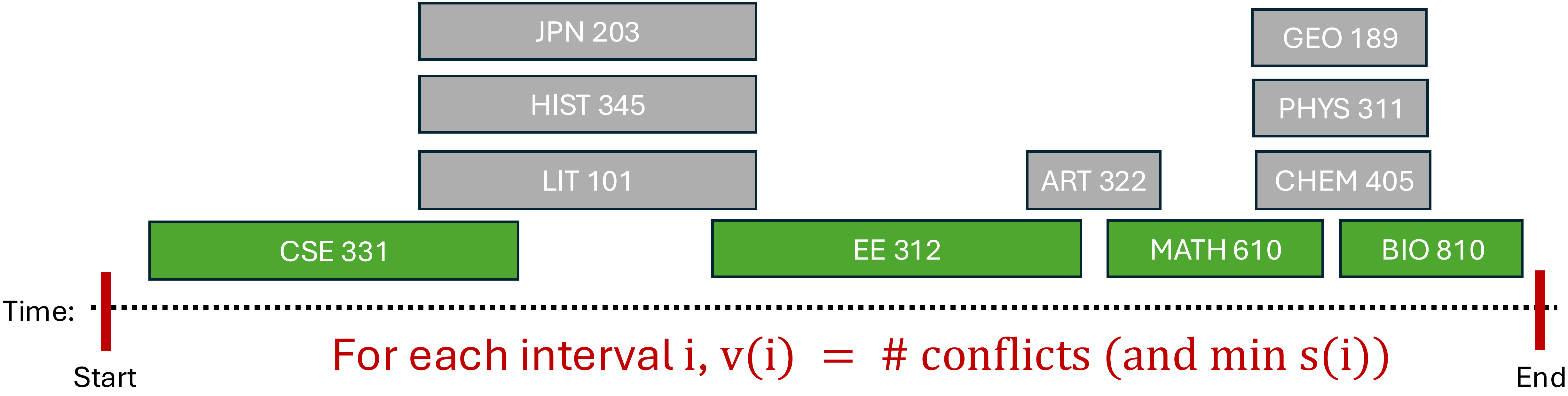
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



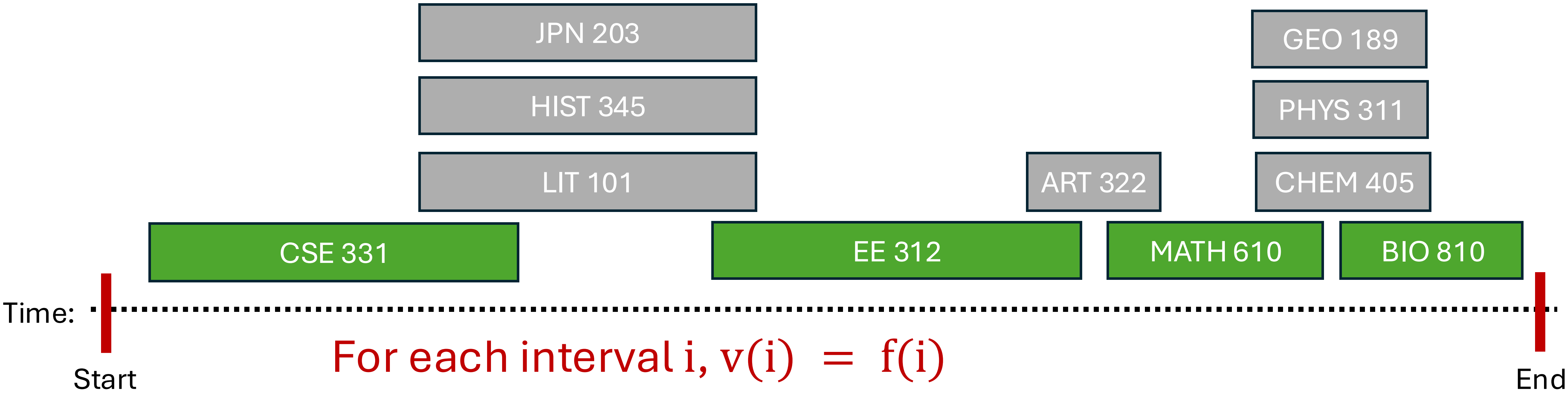
For each interval i, v(i) = # conflicts (and min s(i))

I GIVE UP!!!!



For each interval i, v(i) = # conflicts (and min s(i))

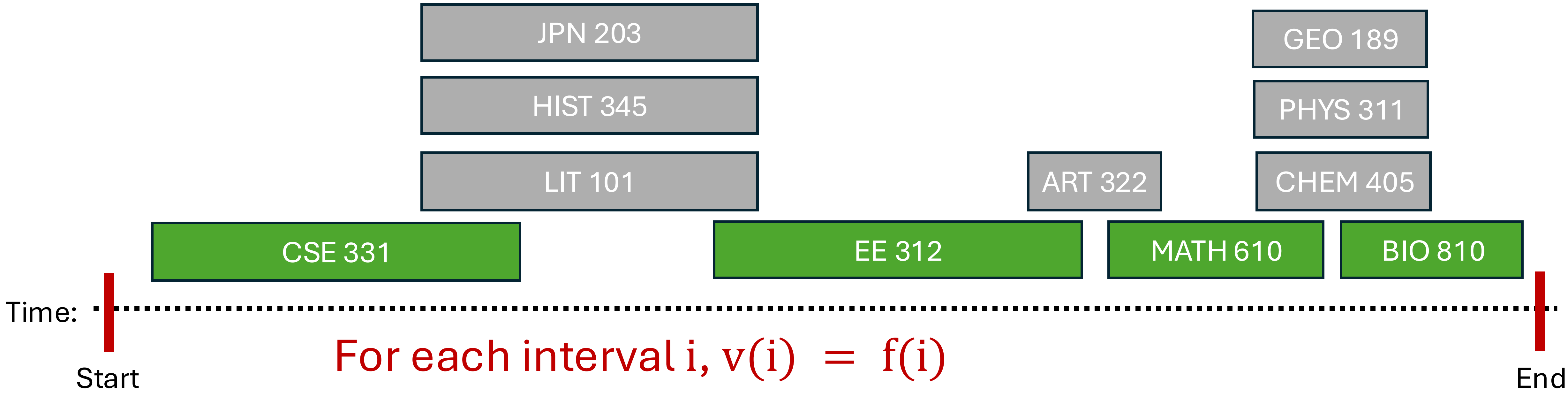
Attempt IV: Finish Time (Okay)



For each interval i, $v(i) = f(i)$

Attempt IV: Finish Time (....)

Wait... does that actually work?

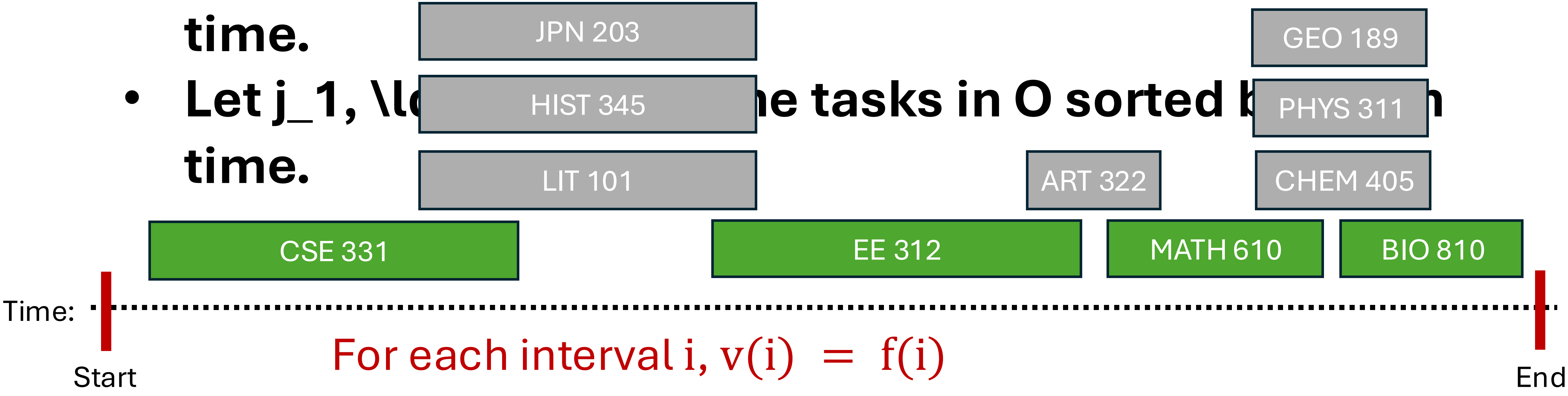


For each interval i, $v(i) = f(i)$

Claim: The Finish First Algorithm is Optimal

Proof Ideas:

- **Let A be the set returned by the algorithm and O be the optimal list.**
- **Let i_1, \ldots, i_k be the tasks in A sorted by add time.**
- **Let j_1, \l[...]e tasks in O sorted b[...]n time.**



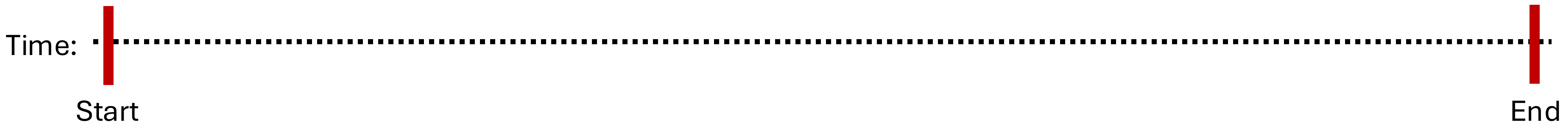
For each interval i, $v(i) = f(i)$

Claim: The Finish First Algorithm is Optimal

Proof Ideas:

- Let $i_1, \ldots, i_k$ be the tasks returned by algorithm (sorted by finish/add time).
- Let $j_1, \ldots, j_m$ be the tasks in optimal solution (sorted by finish time)
- We want to show k = m

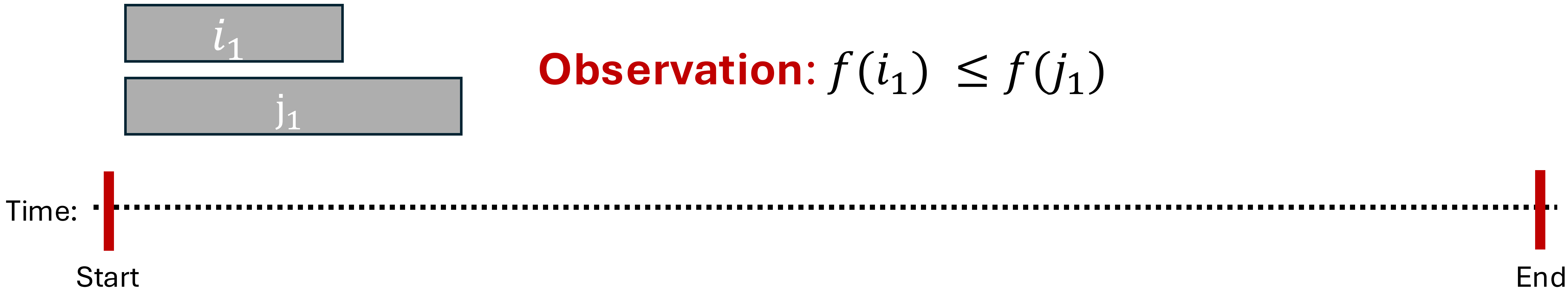
Q: What can we say about the first job in each list?



Claim: The Finish First Algorithm is Optimal

Proof Ideas:

- Let $i_1, \ldots, i_k$ be the tasks returned by algorithm (sorted by finish/add time).
- Let $\mathrm{j}_1, \ldots, \mathrm{j}_\mathrm{m}$ be the tasks in optimal solution (sorted by finish time)



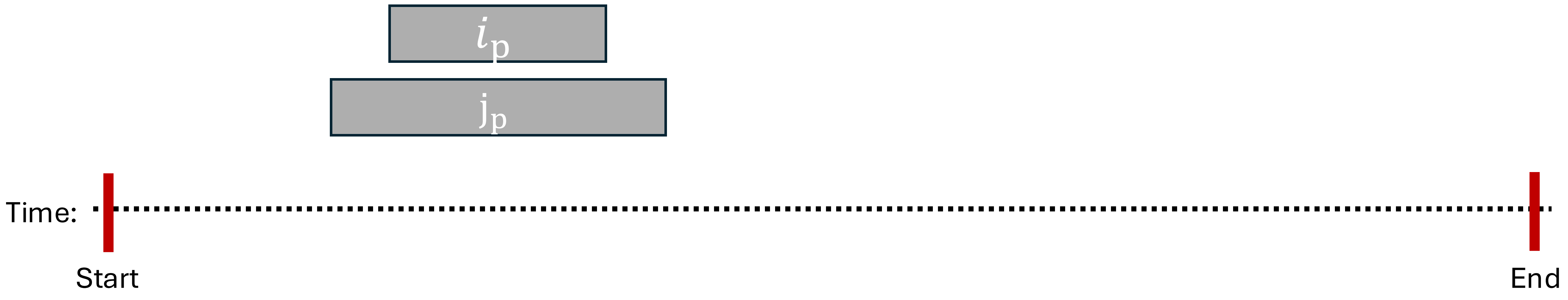
Observation: $f(i_1) \leq f(j_1)$

Claim: The Finish First Algorithm is Optimal

Proof Ideas:

- Assume now that $f(i_p) \leq f(j_p)$ for some p.
  - That is, assume the pth in the algorithms list ends before the pth job in the optimal list.

Q: What can we say about the $\mathrm{p}+1$ job in each list?

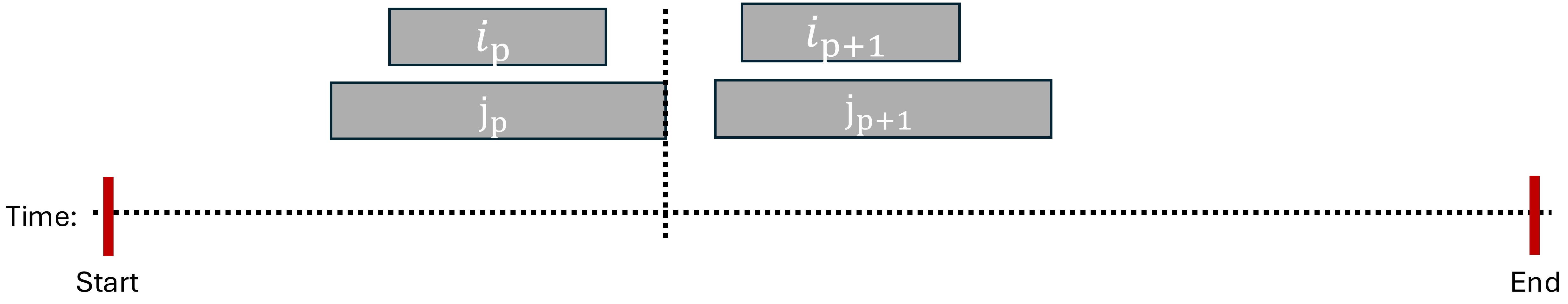


Claim: The Finish First Algorithm is Optimal

Proof Ideas:

- Assume now that $f(i_p) \leq f(j_p)$ for some p.
  - That is, assume the pth in the algorithms list ends before the pth job in the optimal list.

Observation: The algorithm could have added j_{p+1}!

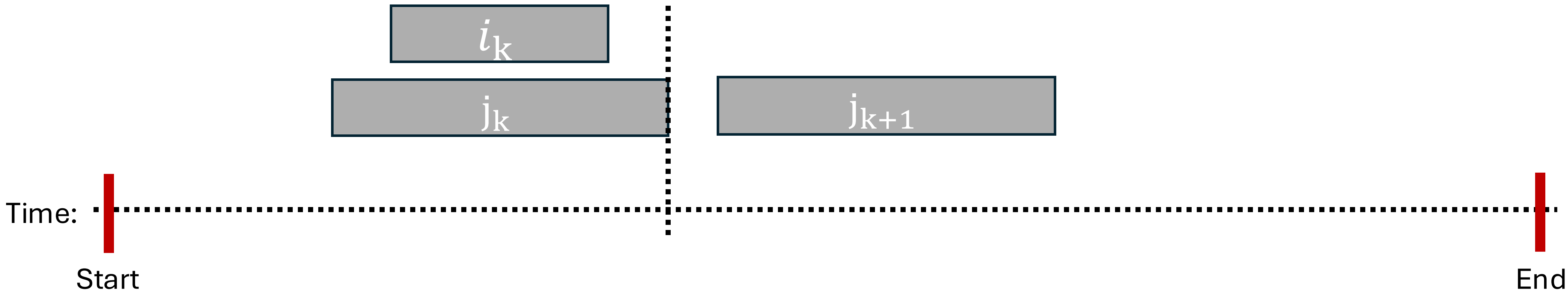


Claim: The Finish First Algorithm is Optimal

Proof Ideas:

- Assume now that $f(i_k) \leq f(j_k)$ and m > k.

Q: Why is this a problem?



Claim: The Finish First Algorithm is Optimal

Proof Ideas:

- Assume now that $f(i_k) \leq f(j_k)$ and m > k.

Observation: The algorithm could have added j_{k+1}!