CSE 331: Algorithms & Complexity

"Greedy Algorithms II"

Prof. Charlie Anne Carlson (She/Her)

Lecture 16

Wednesday October 3rd, 2025

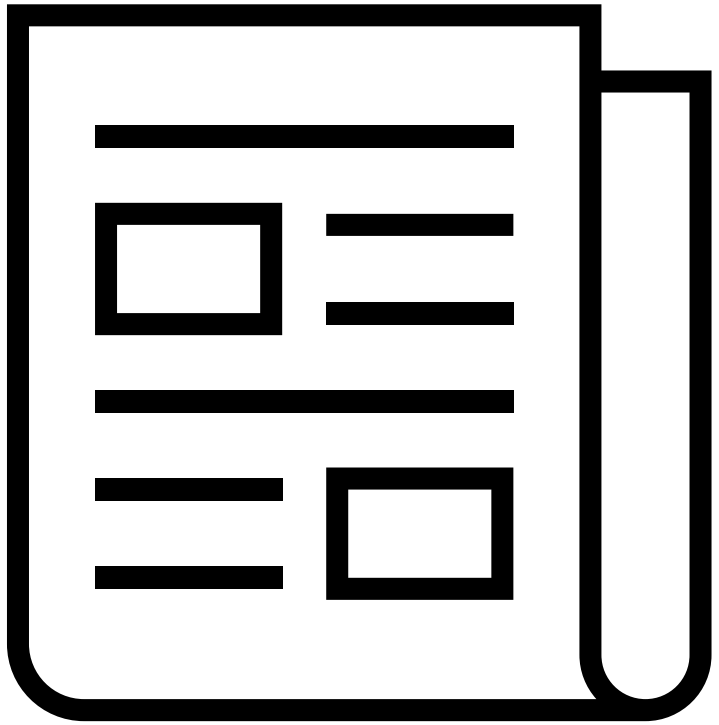
University at Buffalo®

Schedule

1. Course Updates
2. Greedy Algorithms
3. Interval Scheduling
4. Greedy Algorithm
5. Runtime Analysis

Course Updates

- HW 3 & Quiz #1 Grading
- HW 4 Out Soon
  - Not Due Next Week!
- Group Project
  - First Problems Oct 31st
- Sample Midterms Out
- Midterms Oct 6 and Oct 8



Midterm Advice

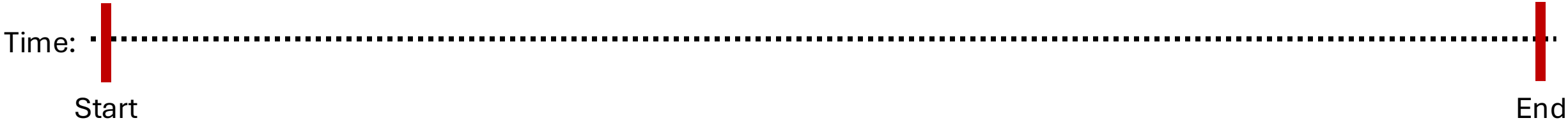
- Midterms Oct 6 and Oct 8.
- You get a reference sheet.
- Exams are in class during regular class time.
- Don’t panic!
  - Sleep, eat, drink water, and study what you can even it that isn’t everything!

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!

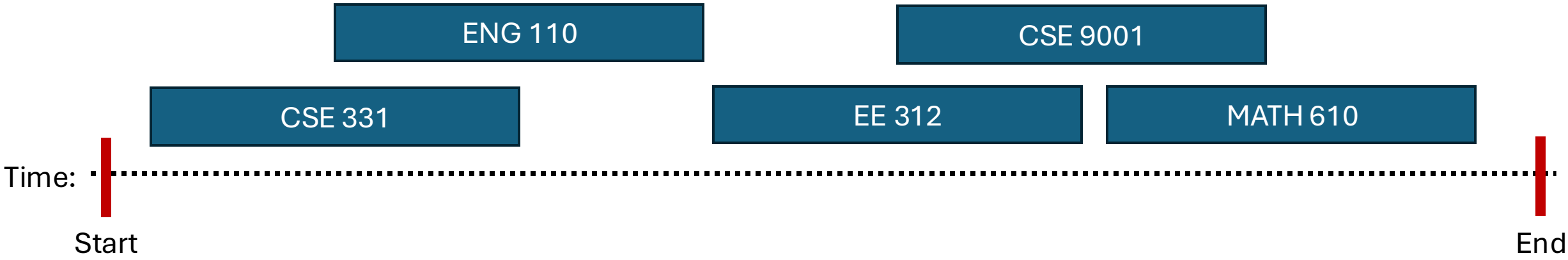
Interval Scheduling

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



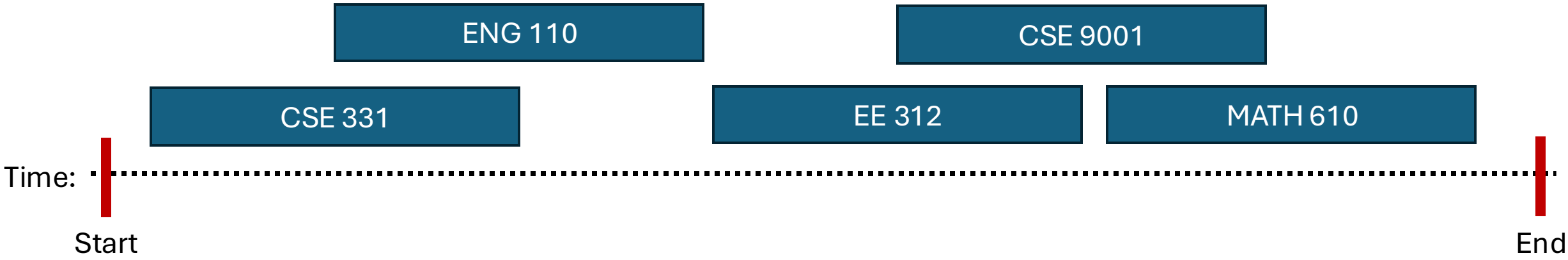
Interval Scheduling

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



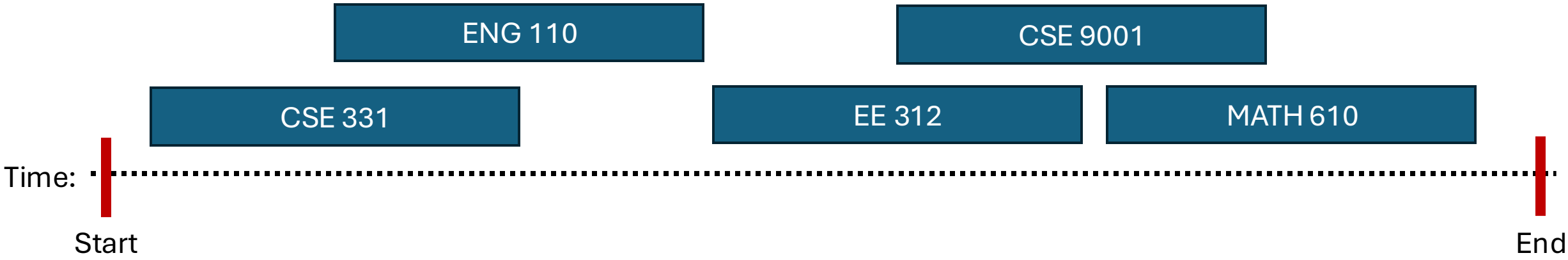
Interval Scheduling

- 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

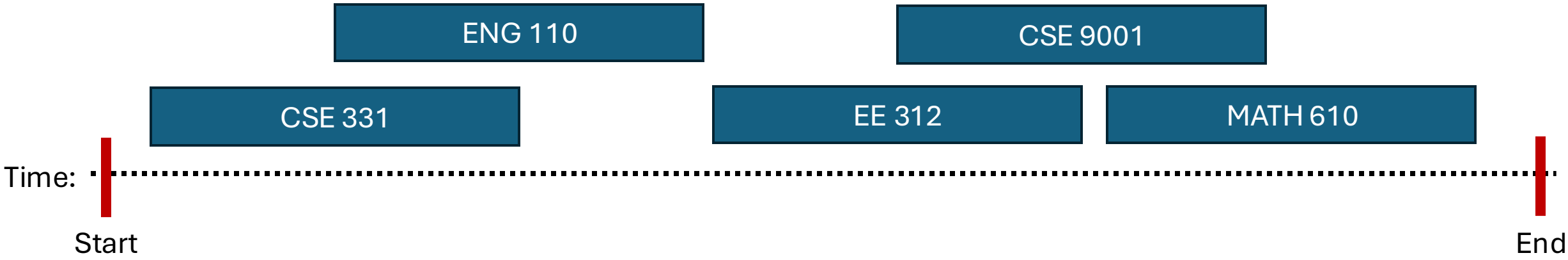
- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.



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.

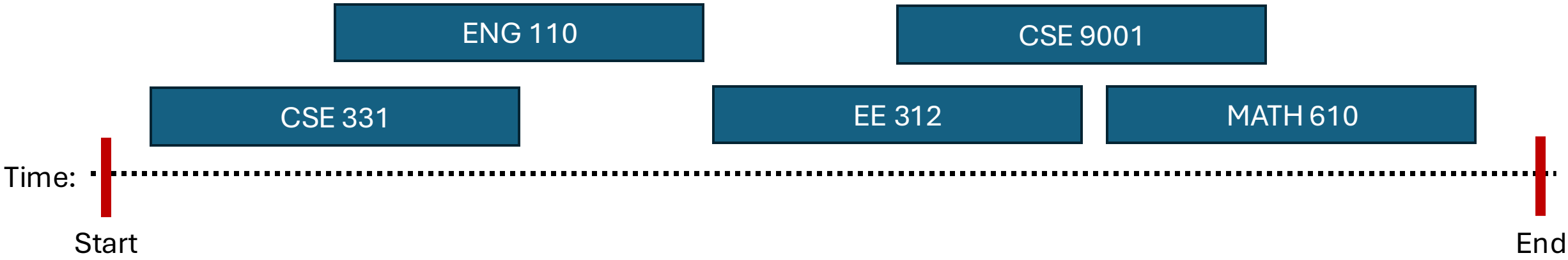
Q: Do we ever pick CSE 9001?



Interval Scheduling

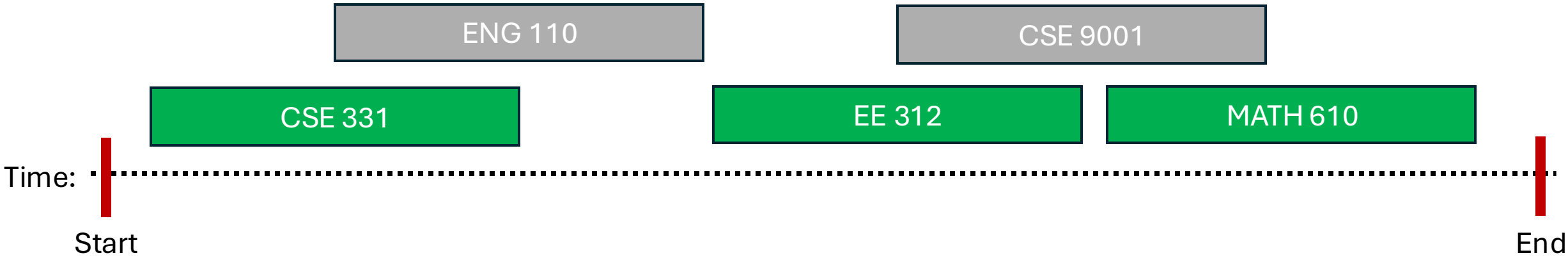
- Consider an interval of time (e.g. Wednesday).
- Consider tasks that need to be completed during specific times (e.g. classes).
- We want to fit as many tasks as possible into the day such that no two overlap.

A: No, because it blocks two classes!



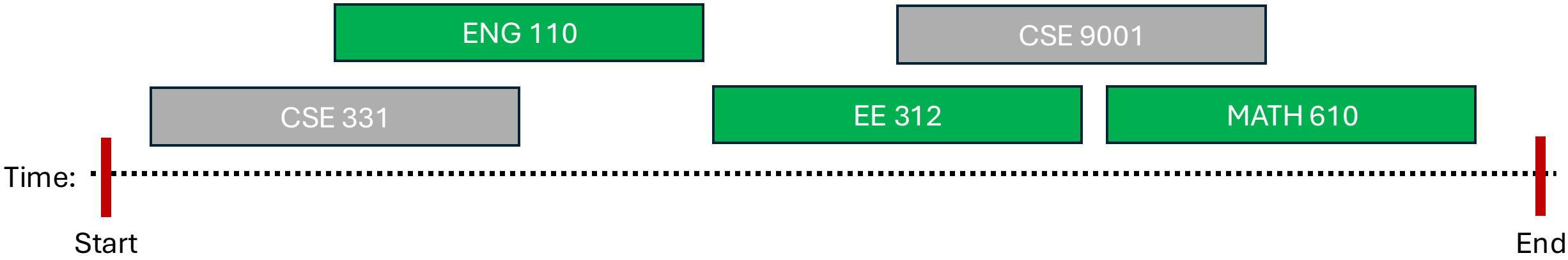
Optimal Solution #1

- 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.

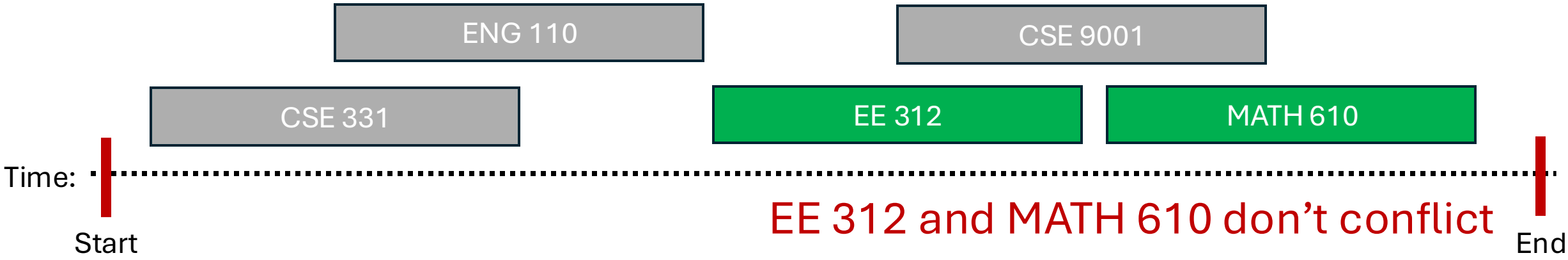
The problem

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.

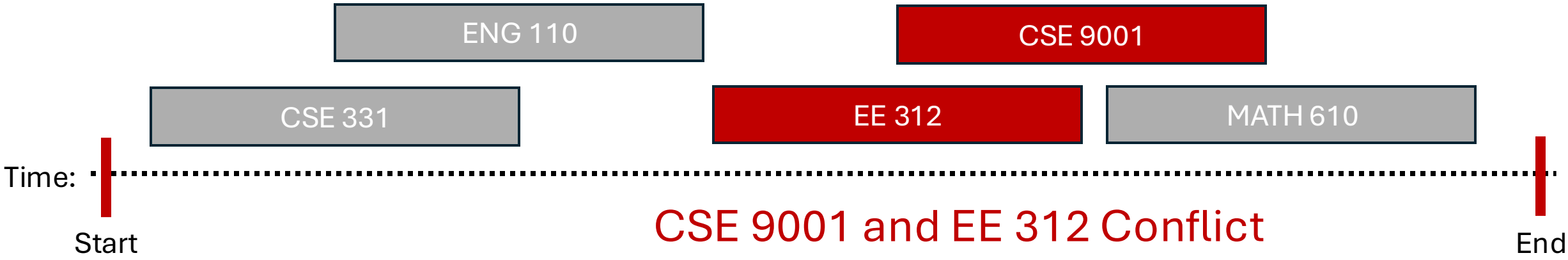
Interval Scheduling Problem

- **Input**: A set of n intervals with start and finish times.
  - For $1 \leq i \leq n$, $[s(i), f(i))$ where $s(i)$ and $f(i)$ 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.



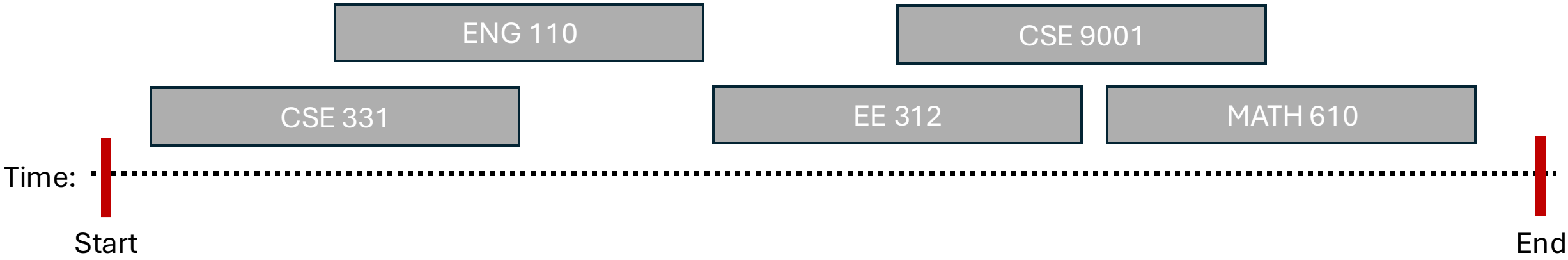
Interval Scheduling Problem

- **Input**: A set of n intervals with start and finish times.
  - For $1 \leq i \leq n$, $[s(i), f(i))$ where $s(i)$ and $f(i)$ 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 $1 \leq i \leq n$, $[s(i), f(i))$ where $s(i)$ and $f(i)$ 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?

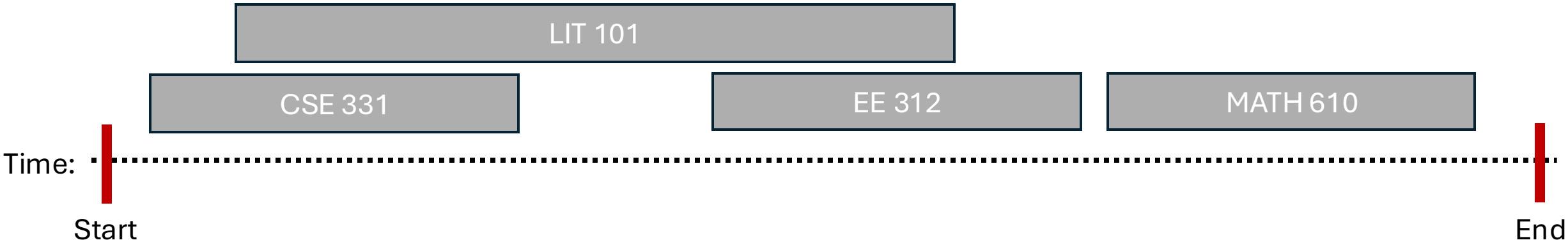
A: Let’s try to generate some examples and see what works and what doesn’t.

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

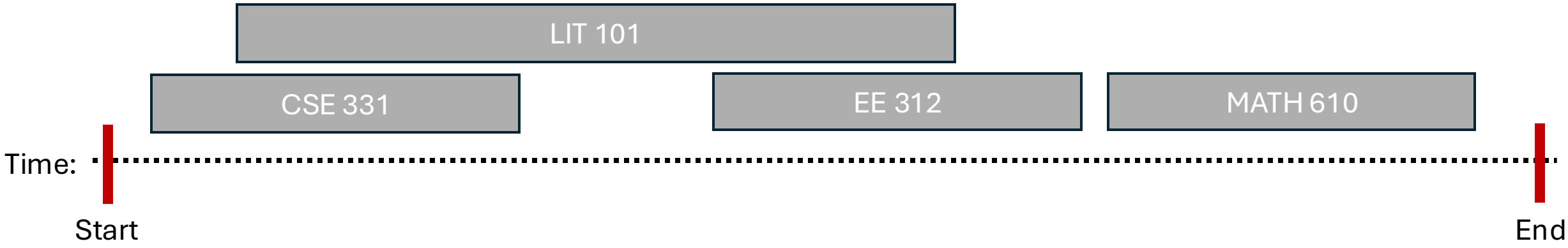
Build an Algorithm

- 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



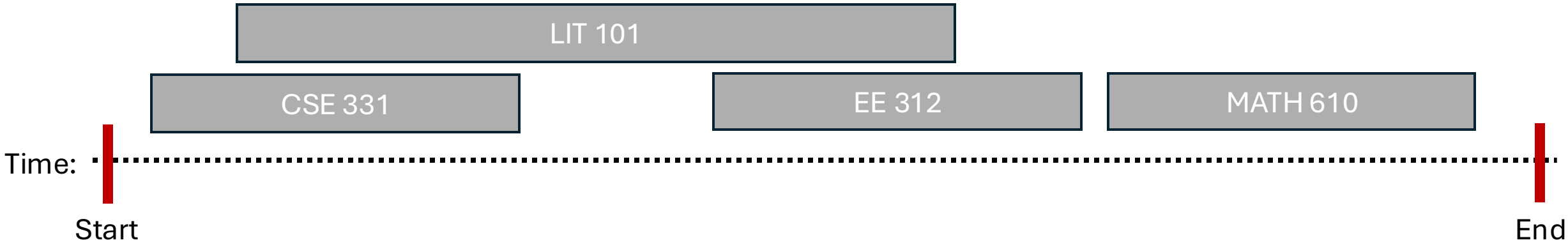
Build an Algorithm

- 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



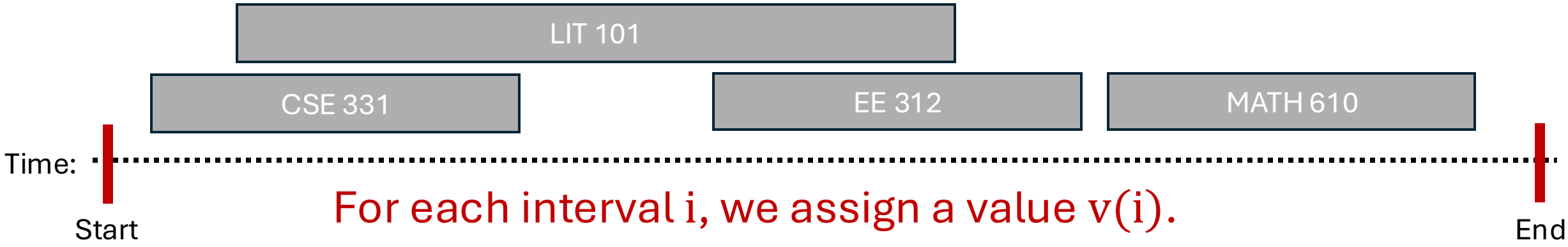
Build an Algorithm

- 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

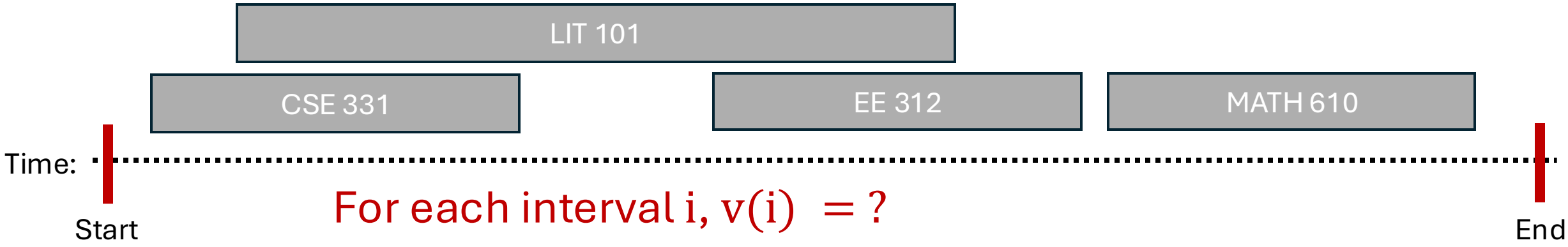
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove **all tasks that conflict with** i from R



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

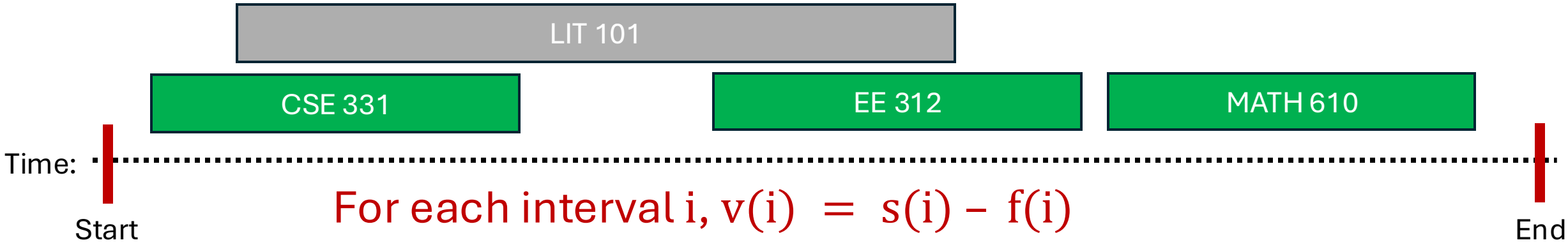
Q: What should we pick for $v(i)$?

- 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)

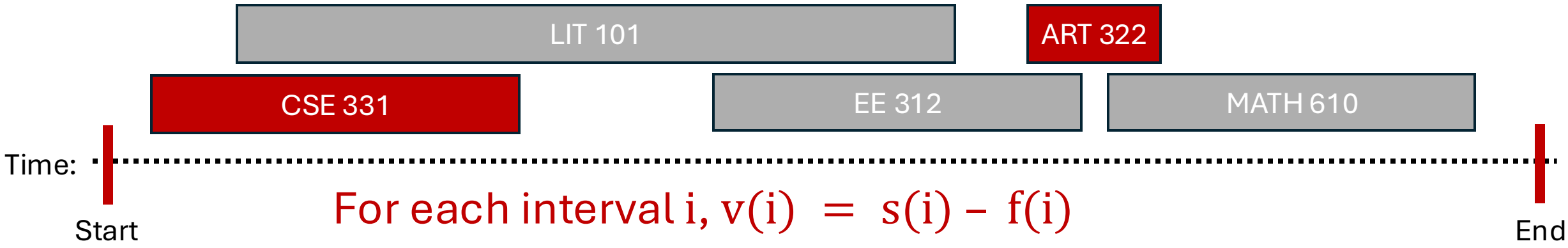
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



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

Attempt I: Interval Length (Oh no!)

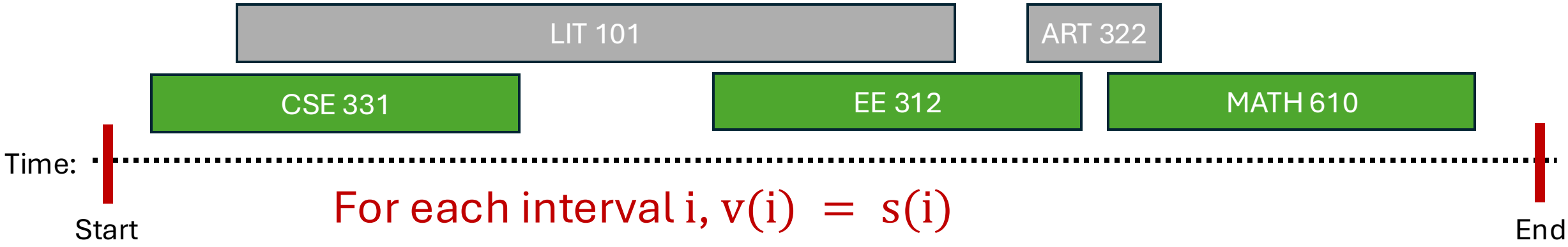
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



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

Attempt II: Start Time (Okay)

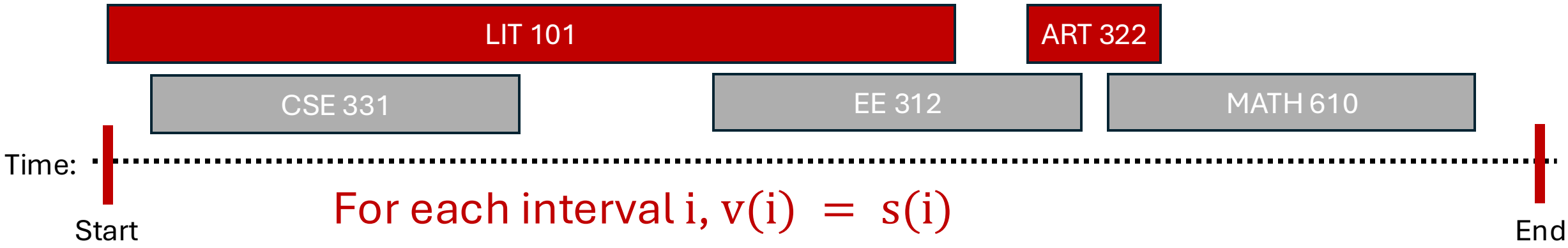
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



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

Attempt II: Start Time (Oh no!)

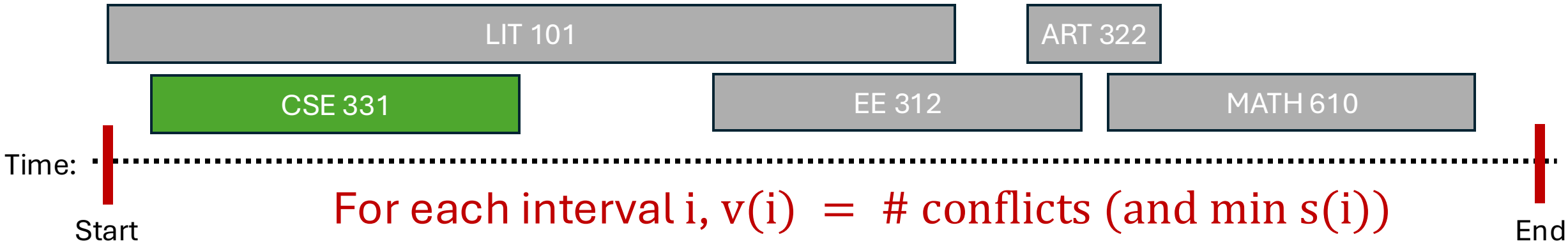
- Basic Algorithm Outline:
  - S is empty
  - While R is not empty:
    - Pick i in R that minimizes v(i)
    - Add i to S
    - Remove all tasks that conflict with i from R



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

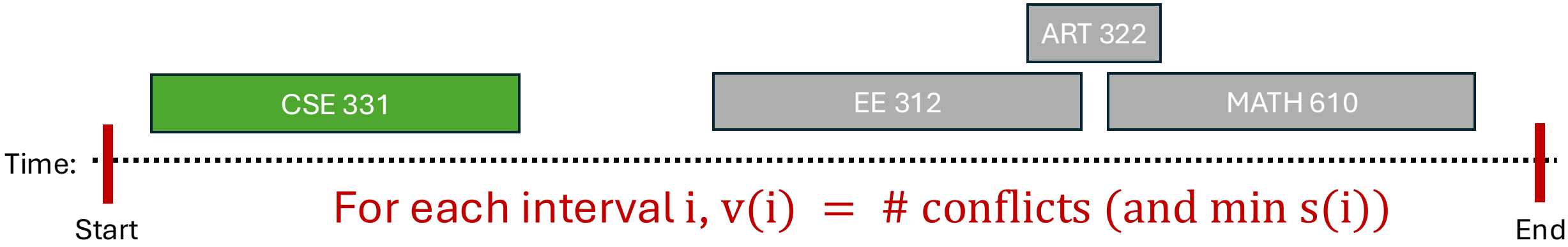
Attempt III: Min Conflicts (Okay)

- 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



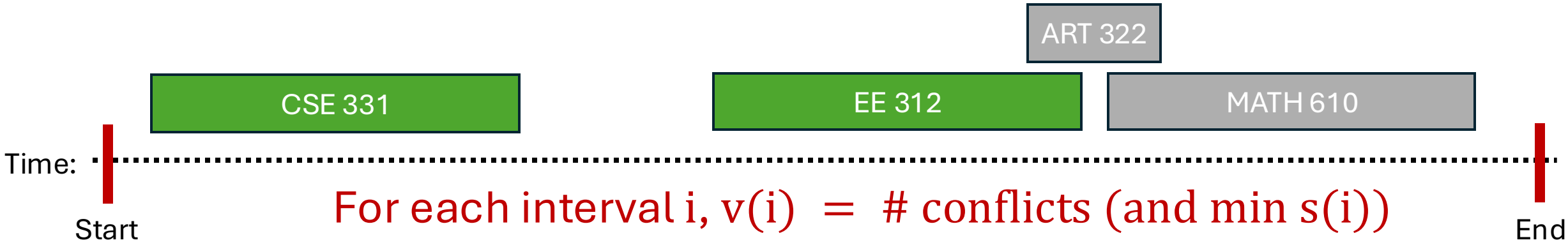
Attempt III: Min Conflicts (Okay)

- 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



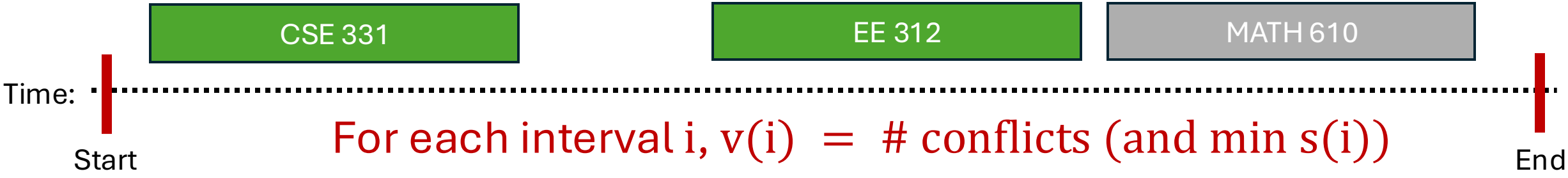
Attempt III: Min Conflicts (Okay)

- 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



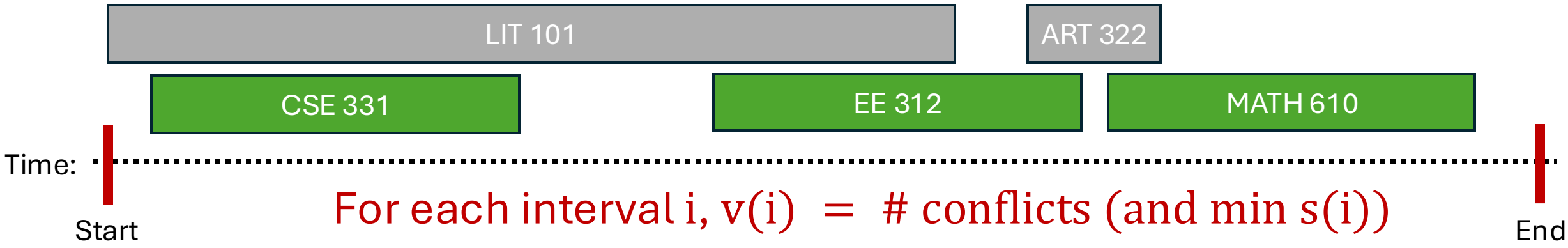
Attempt III: Min Conflicts (Okay)

- 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



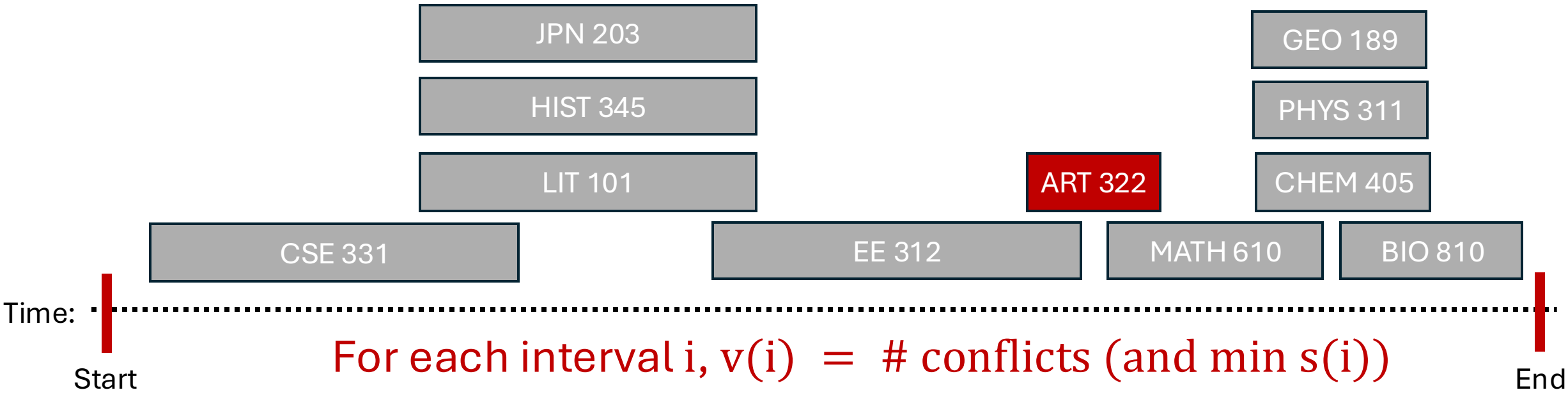
Attempt III: Min Conflicts (Okay)

- 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



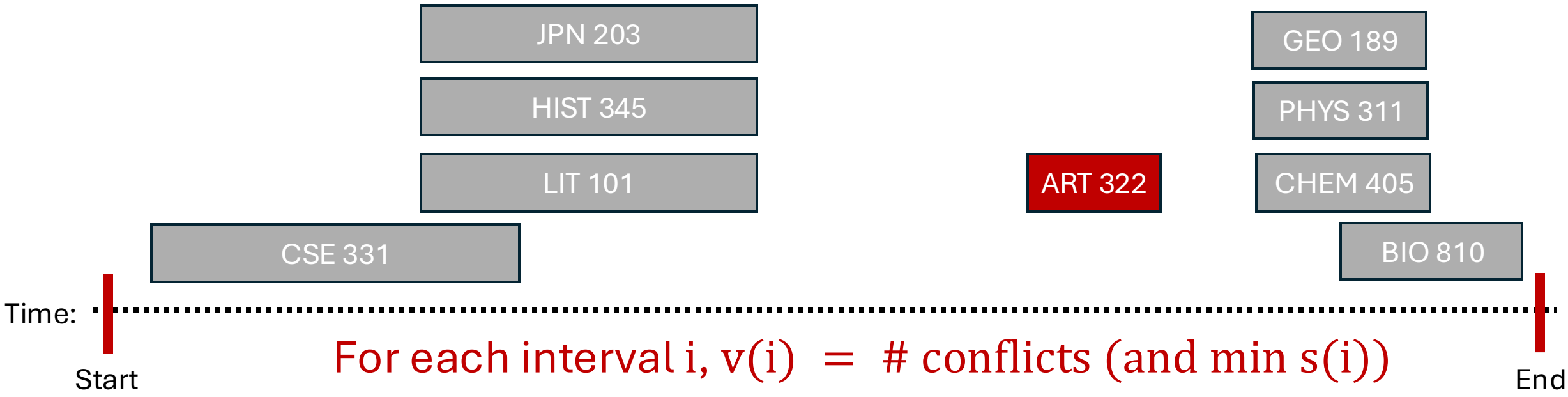
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



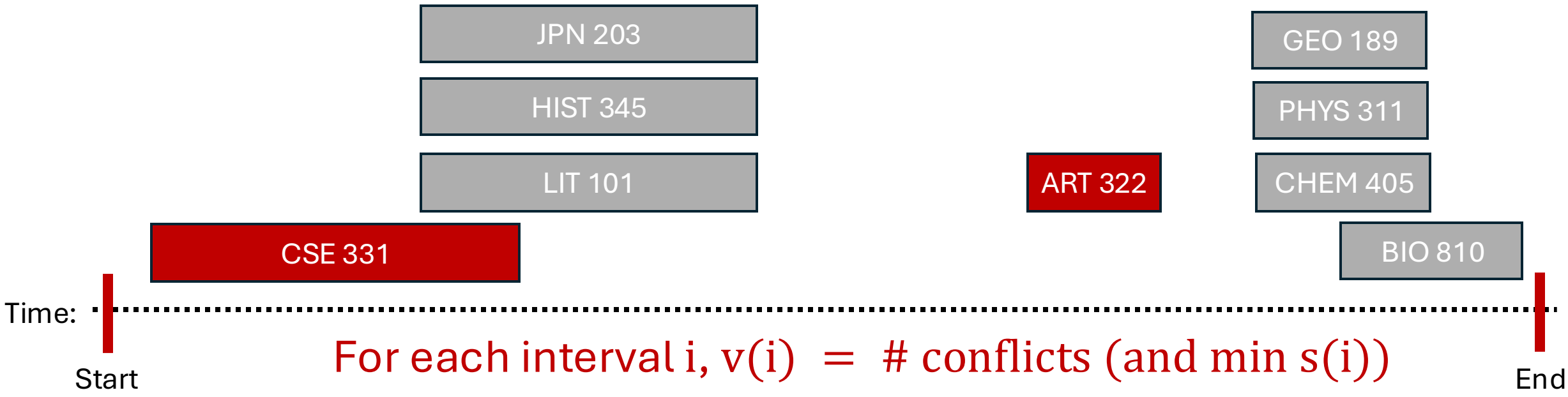
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



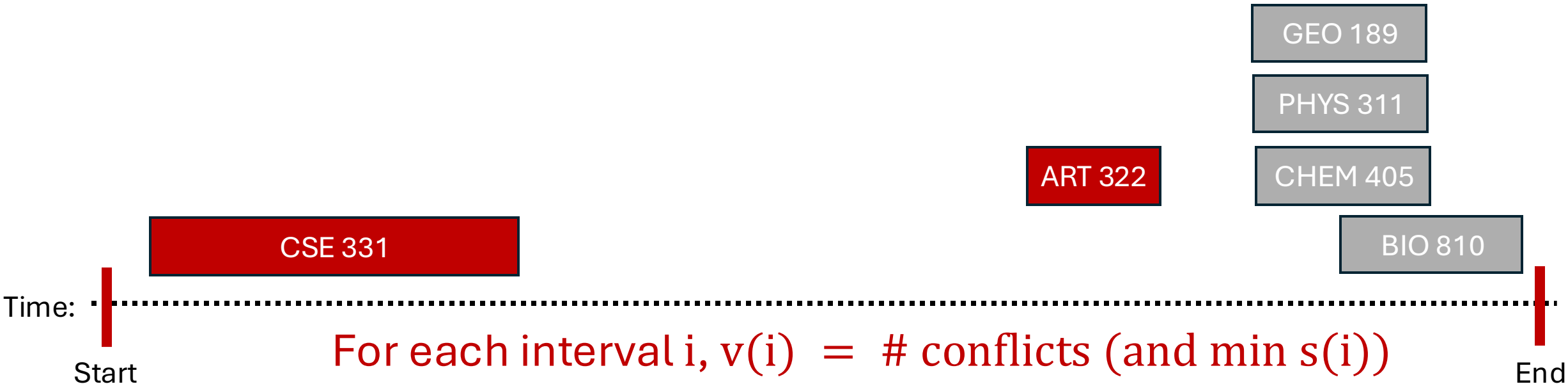
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



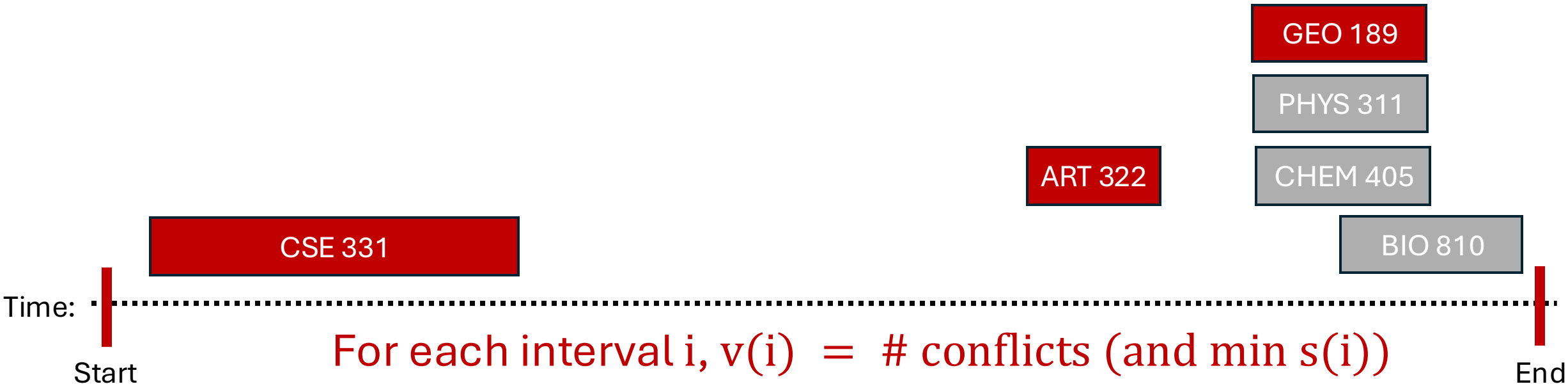
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



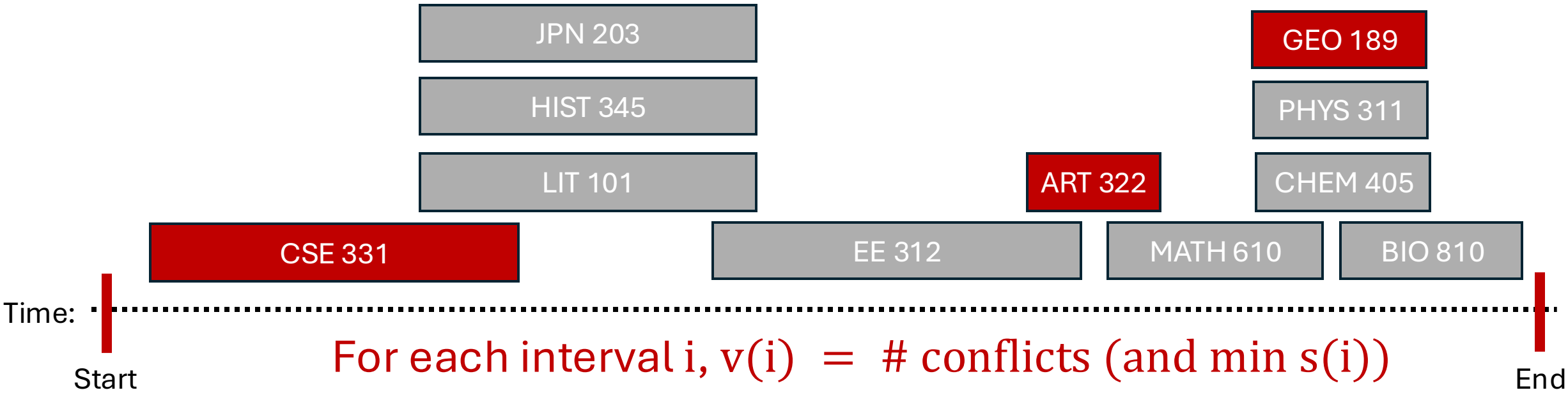
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



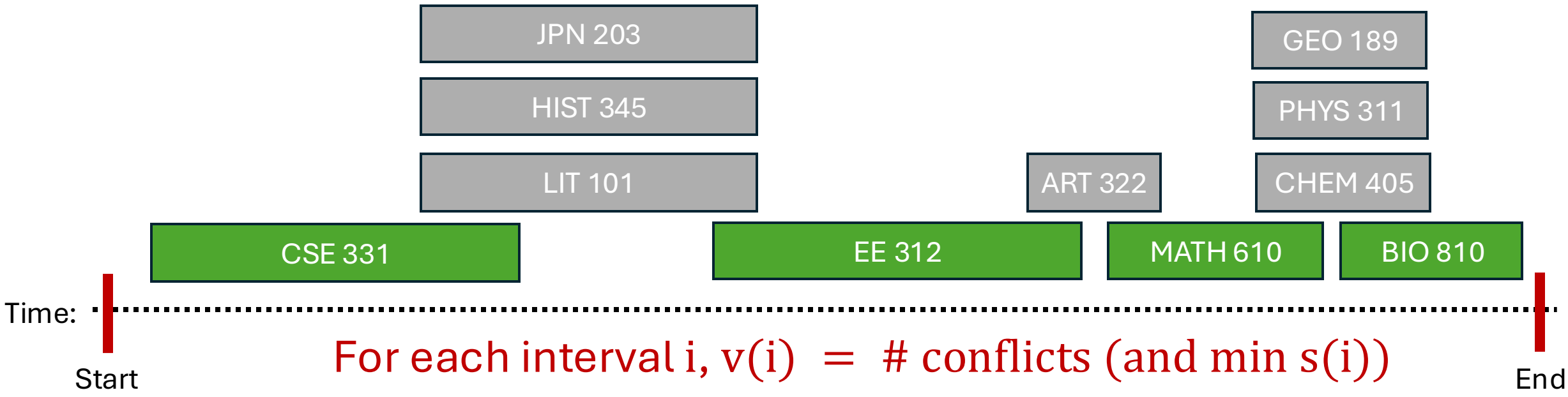
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



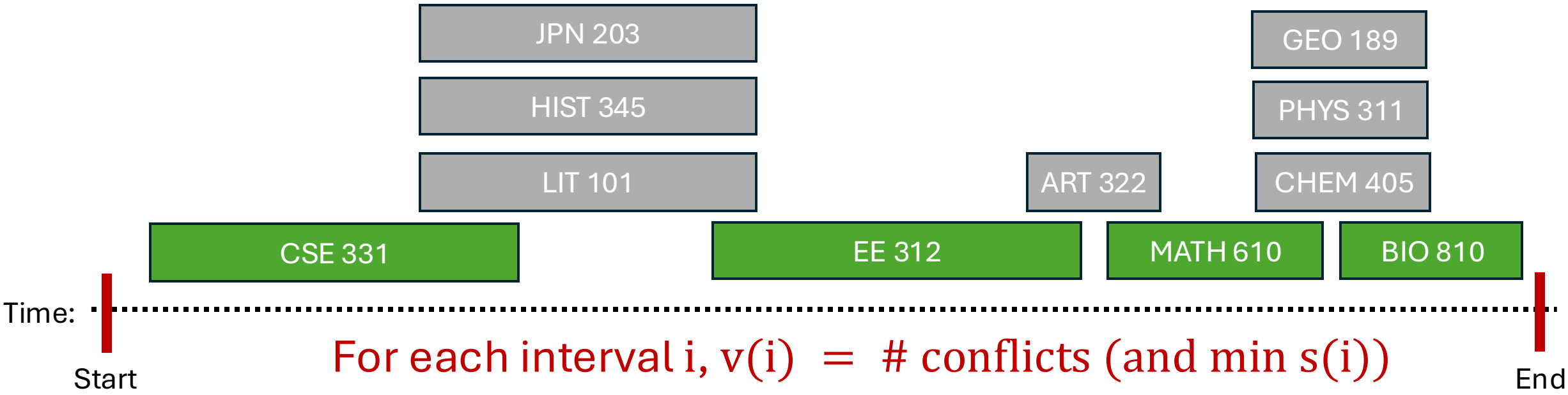
For each interval i, v(i) = # conflicts (and min s(i))

Attempt III: Min Conflicts (Oh no!)



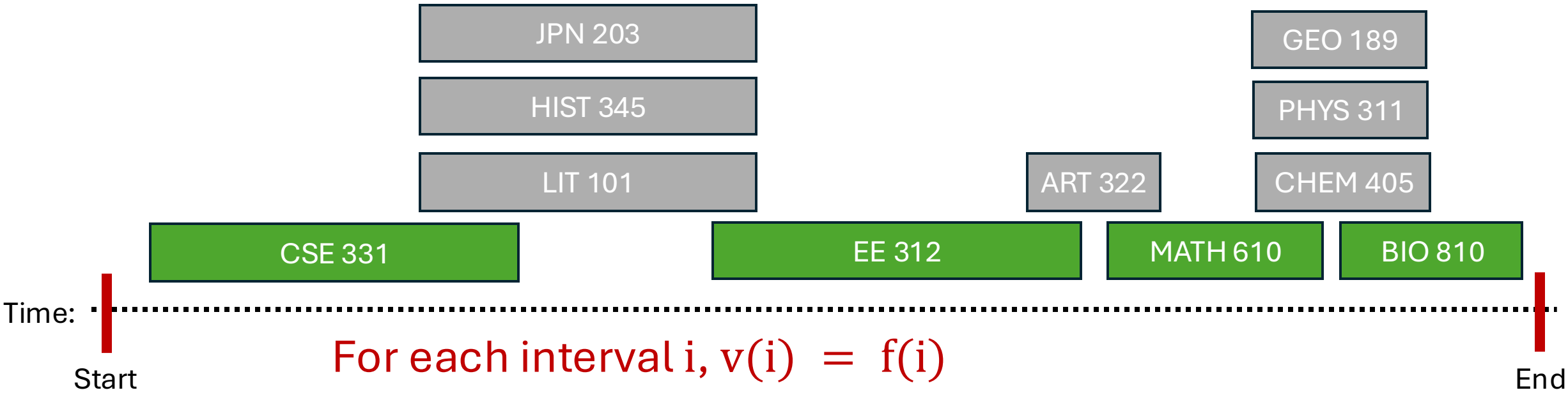
For each interval i, v(i) = # conflicts (and min s(i))

I GIVE UP!!!!



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

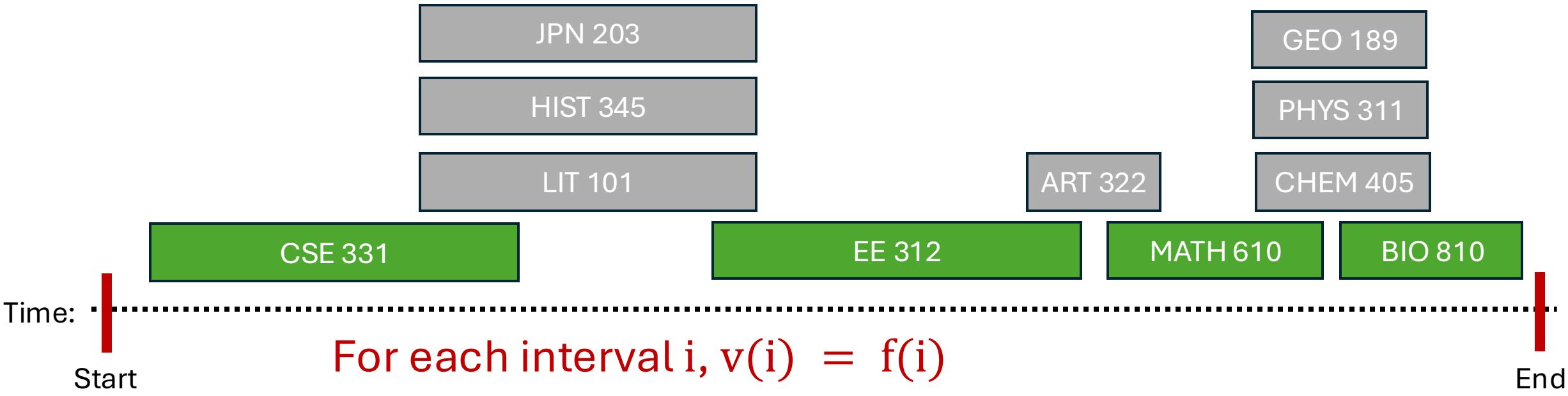
Attempt IV: Finish Time (Okay)



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

Attempt IV: Finish Time (….)

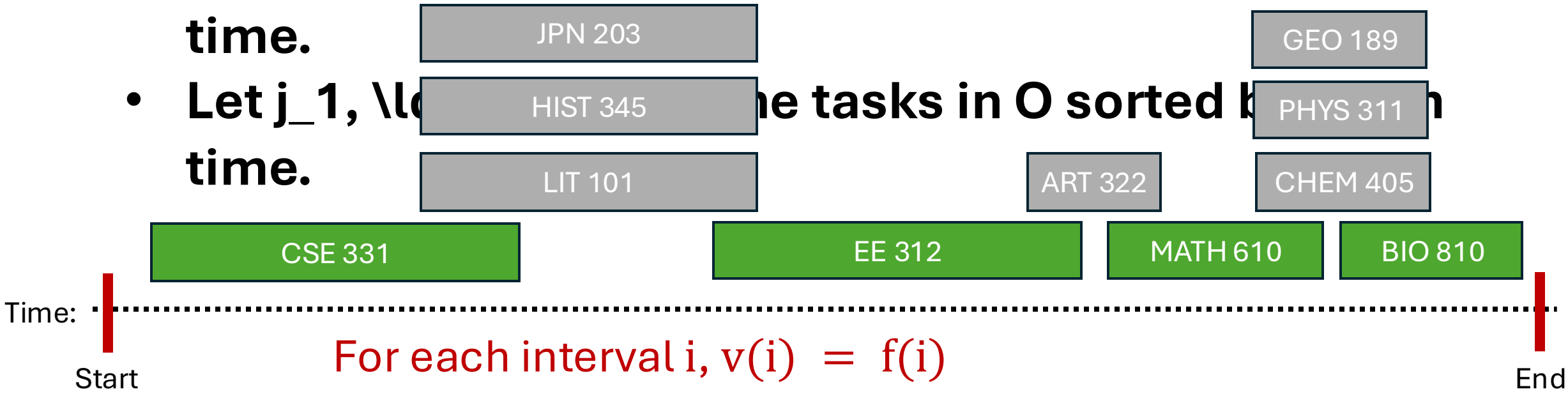
Wait… does that actually work?



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[illegible]e tasks in O sorted b[illegible]n time.**



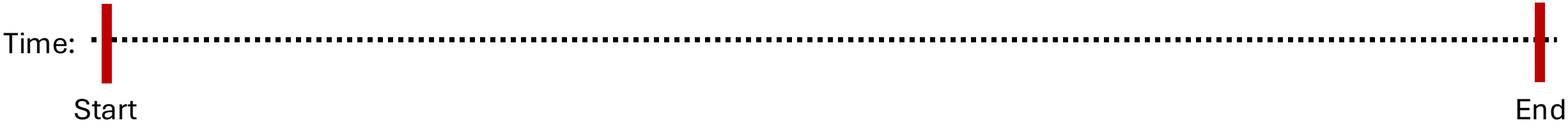
For each interval i, $v(i) = f(i)$

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

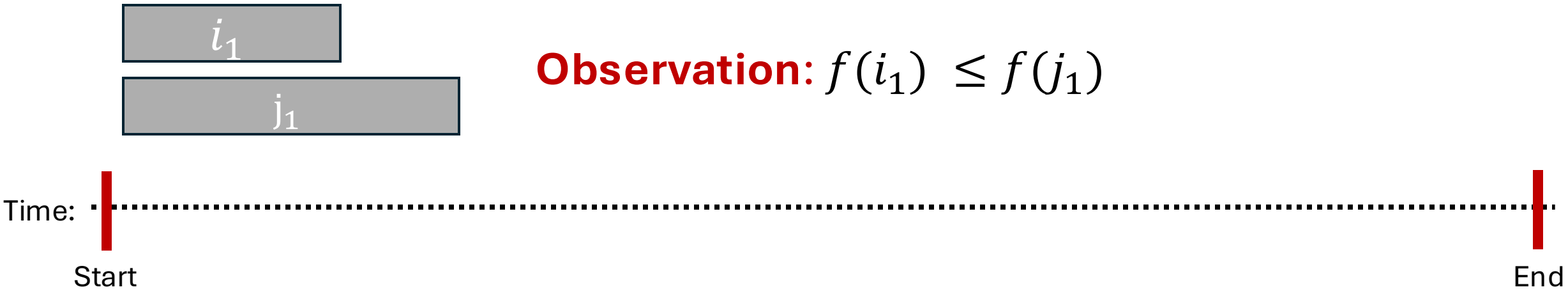
Q: What can we say about the first job in each list?



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 $j_1, \ldots, j_m$ be the tasks in optimal solution (sorted by finish time)



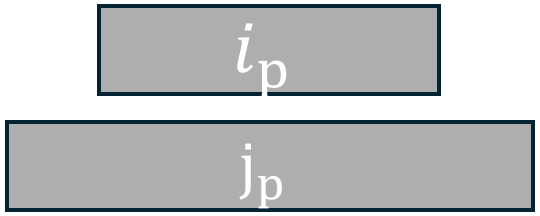
Observation: $f(i_1) \leq f(j_1)$

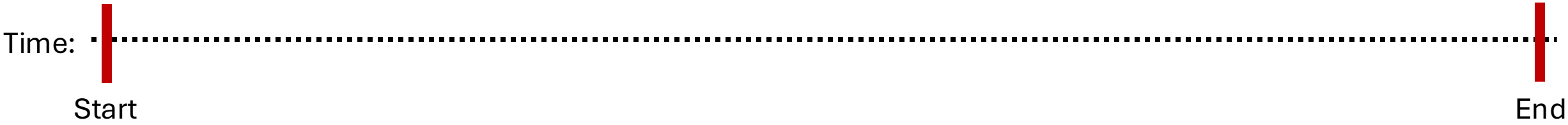
Claim: The Finish First Algorithm is Optimal

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 $p + 1$ job in each list?



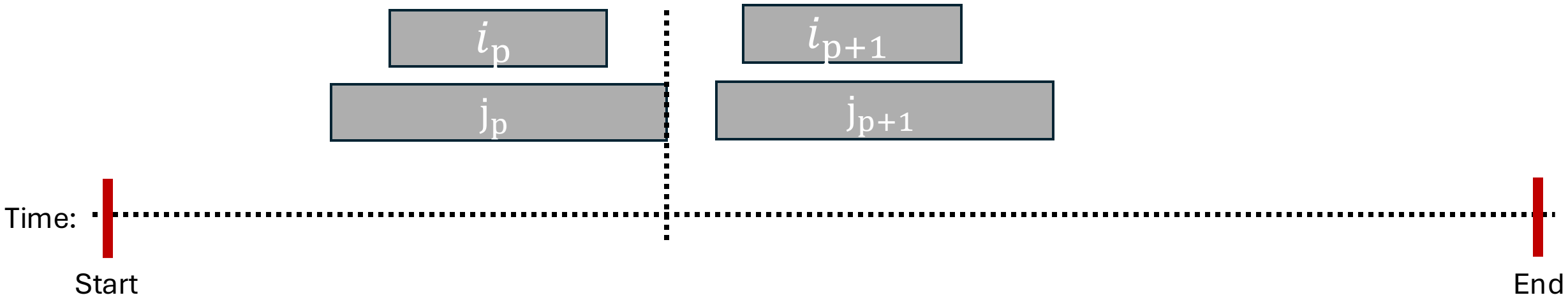


Claim: The Finish First Algorithm is Optimal

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}!

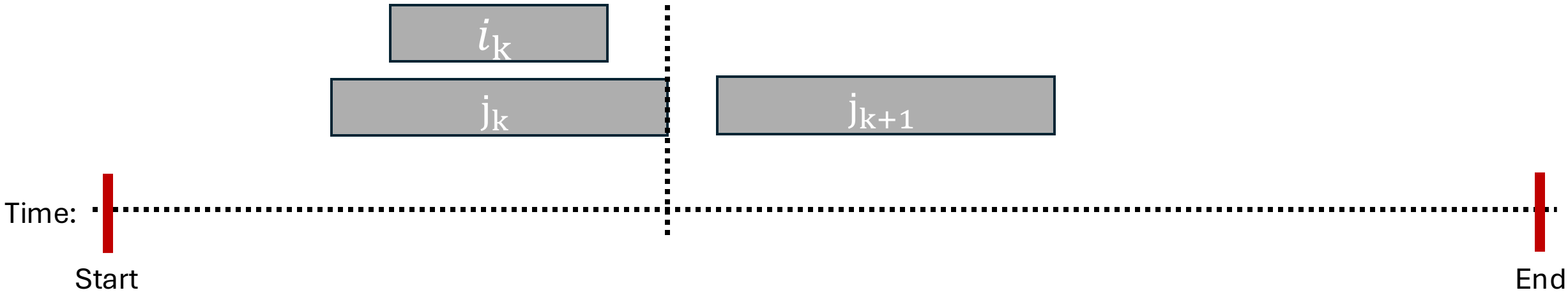


Claim: The Finish First Algorithm is Optimal

Proof Ideas:

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

Q: Why is this a problem?

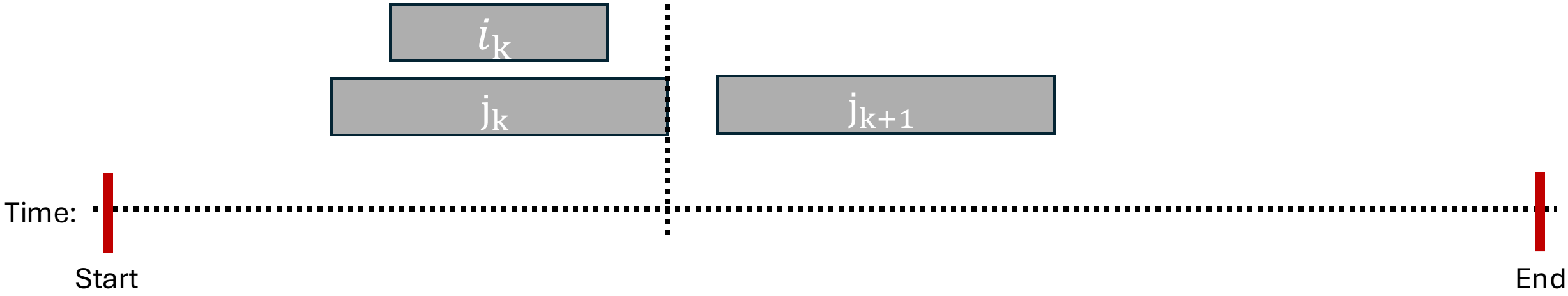


Claim: The Finish First Algorithm is Optimal

Proof Ideas:

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

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



Runtime Analysis

- 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

Runtime Analysis

- **Input**: List of tasks R
  - S is empty
  - While R is not empty:
    - Find i in R with earliest finish time
    - Add i to S
    - Remove all tasks that conflict with i from R
  - Return S

Runtime Analysis

- **Input**: List of tasks R
  - S is empty list **<- O(1) time**
  - While R is not empty: **<- O(n) iterations**
    - Find i in R with earliest finish time **<- O(n) time to search**
    - Add i to S **<- O(1) time**
    - Remove all tasks that conflict with i from R **<- O(n) time**
  - Return S **<- O(1) time**

Runtime Analysis: O(n^2)

- **Input**: List of tasks R
  - S is empty list **<- O(1) time**
  - While R is not empty: **<- O(n) iterations**
    - Find i in R with earliest finish time **<- O(n) time to search**
    - Add i to S **<- O(1) time**
    - Remove all tasks that conflict with i from R **<- O(n) time**
  - Return S **<- O(1) time**

Q: Can we do this faster?

Runtime Analysis: O(nlog(n))

- **Input**: List of tasks R
  - Sort R by finish time **<- O(nlog(n)) time**
  - S is empty list **<- O(1) time**
  - Set last_job_fin to be 0 **<- O(1) time**
  - For j = 1 up to n: **<- O(n) iterations**
    - if s(j) > last_job_fin: **<- O(1) time**
      - Add i to S **<- O(1) time**
      - Set last_job_fin to be f(i) **<- O(1) time**
  - Return S **<- O(1) time**