CSE 331: Algorithms & Complexity

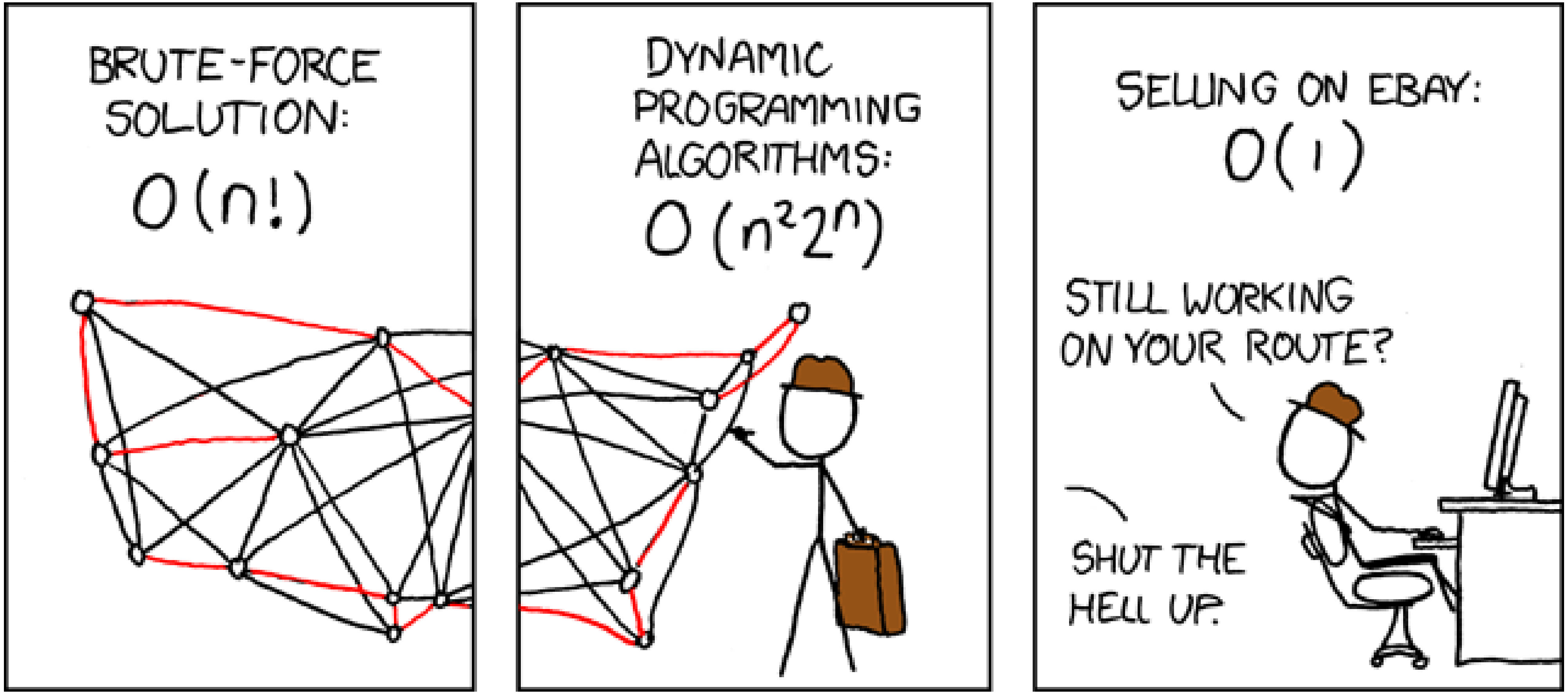
“Dynamic Programming”

Prof. Charlie Anne Carlson (She/Her)

Lecture 28

Monday Nov 7th, 2025

University at Buffalo



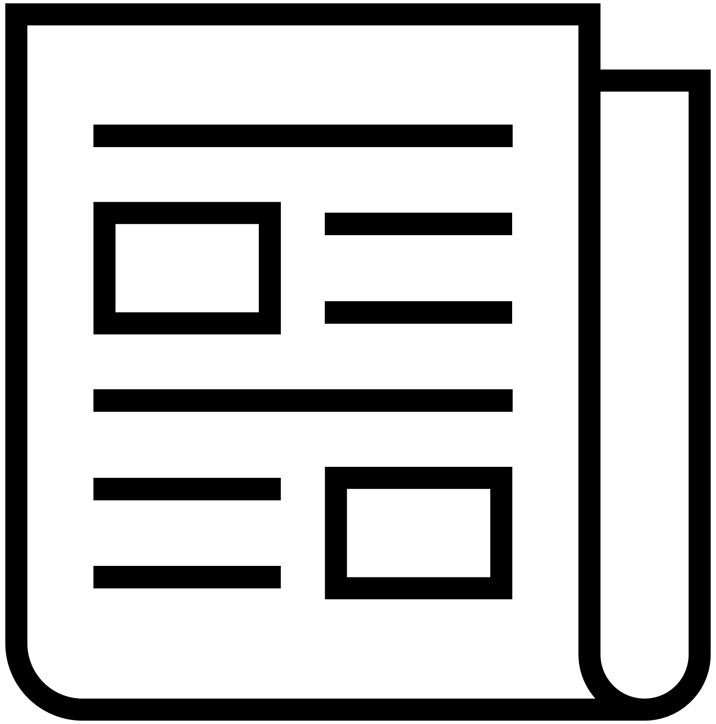
https://xkcd.com/399/

Schedule

1. Course Updates
2. Weighted Interval Scheduling
3. Memoization
4. Dynamic Programming

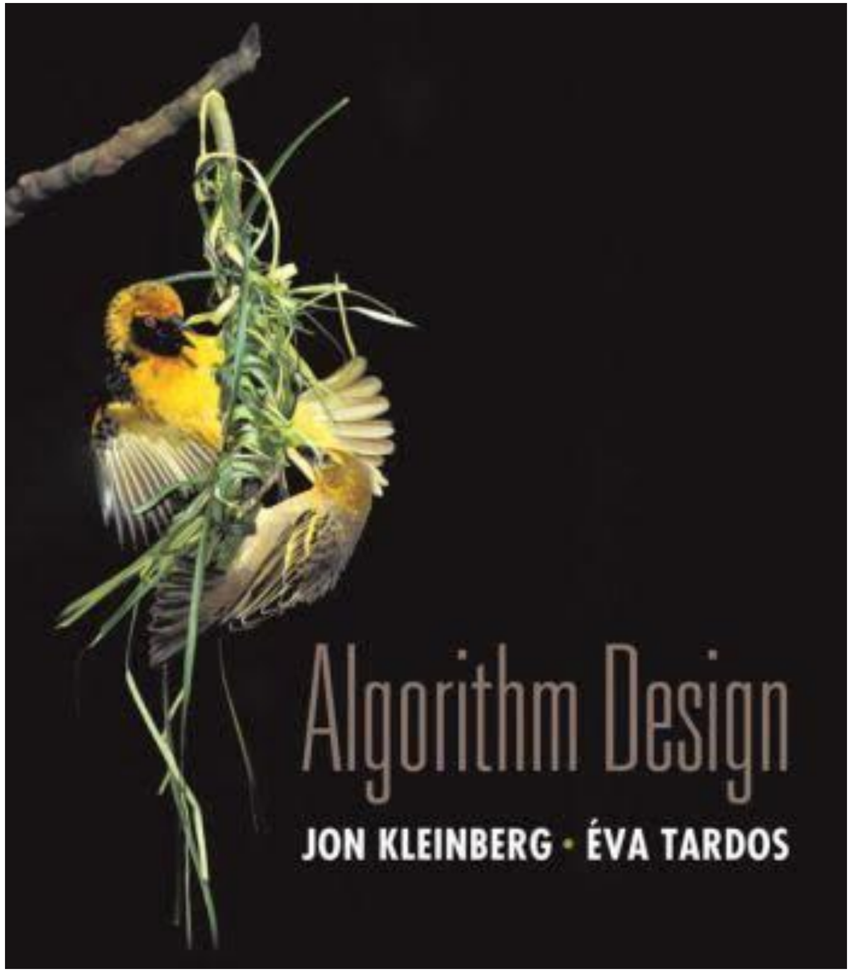
Course Updates

- HW 6 Out
  - Autolab Up
  - Due November 11th
- Group Project
  - Code 3 Due November 24th
  - Reflections 3 Due December 1st
- Check Piazza for Google Form Review Link (before Friday)
- Next Quiz is December 1st



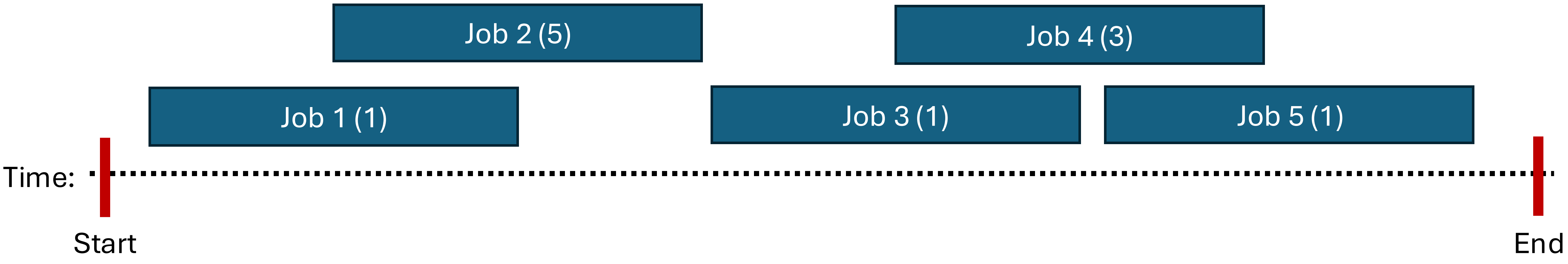
Reading

- You should have read:
  - Started 6.1
  - Started 6.2
- Before Next Class:
  - Finish 6.1
  - Finish 6.2



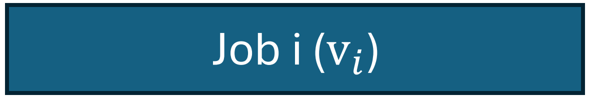
Weighted Interval Scheduling

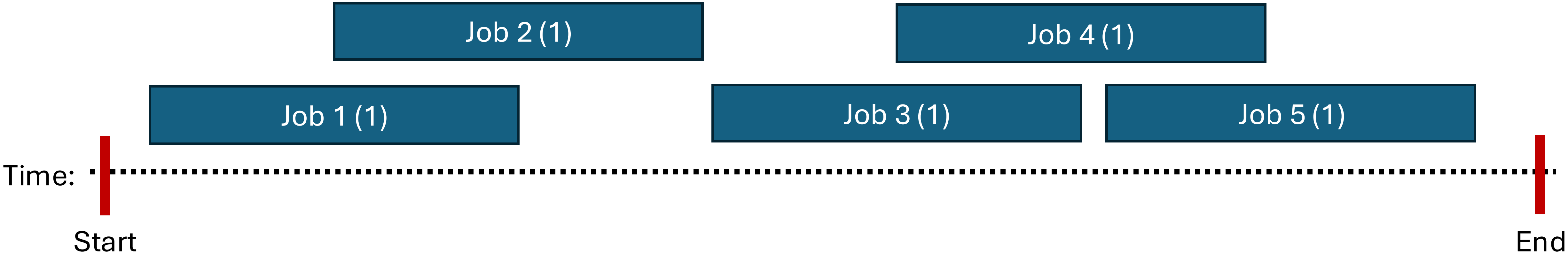
- **Input**: A list of n jobs L
  - Each job i has a start time s_i and finish time f_i
  - Two jobs are "compatible" if they don't overlap
  - Each job i as a weight v_i
- **Goal**: Find the max-weight subset of mutually compatible jobs.



Weighted Interval Scheduling

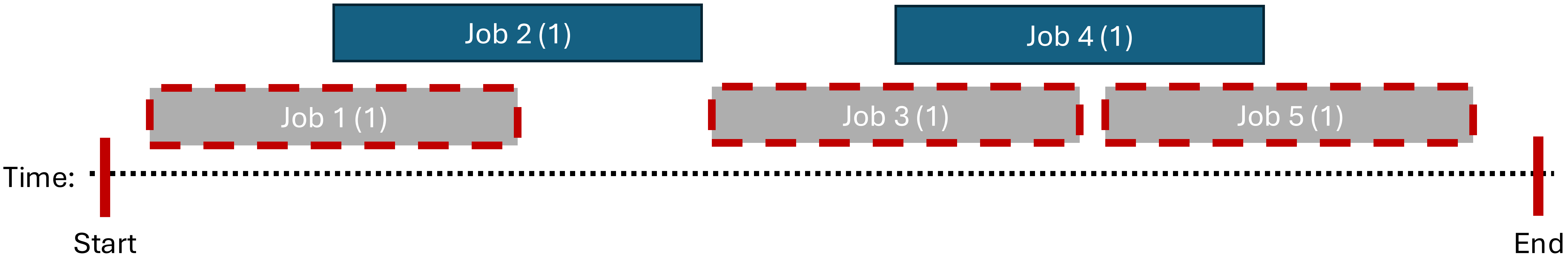
- **Input**: A list of n jobs L
  - Each job i has a start time s_i and finish time f_i
  - Two jobs are "compatible" if they don't overlap
  - Each job i as a weight v_i
- Goal: Find the max-weight subset of mutually compatible jobs.





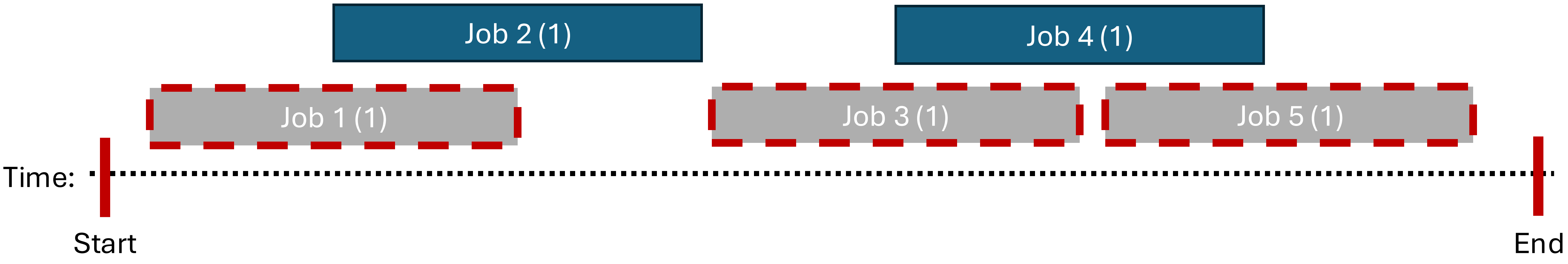
Weighted Interval Scheduling

- **Input**: A list of n jobs L
  - Each job i has a start time s_i and finish time f_i
  - Two jobs are “compatible” if they don’t overlap
  - Each job i as a weight v
- **Goal**: Find the max-weight subset of mutually compatible jobs.



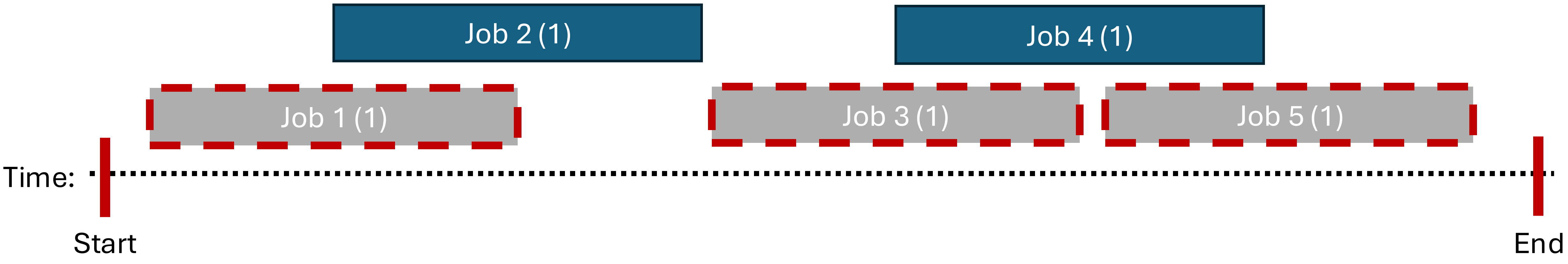
Unweighted Case

- **Question:** What algorithm do we use to find the maximum weight subset when the weight of each job is 1?



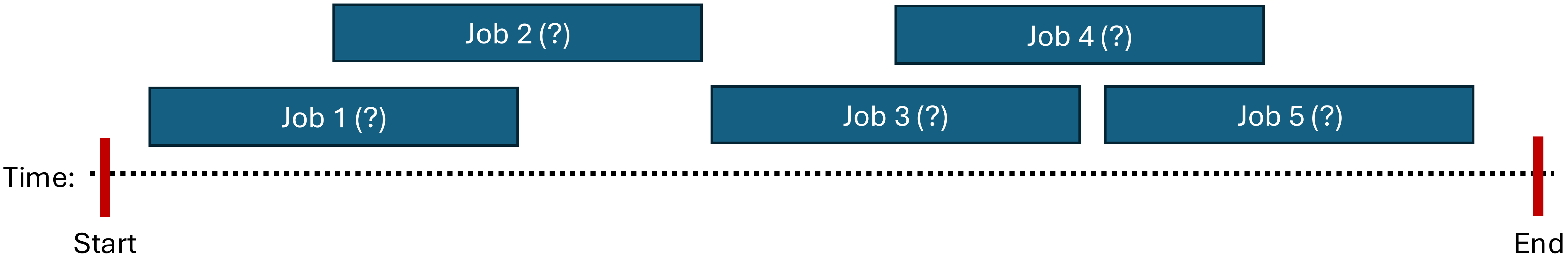
Unweighted Case

- **Question**: What algorithm do we use to find the maximum weight subset when the weight of each job is 1?
- **Answer**: We use a greedy algorithm where we scan from left to right and always take the next job to finish.



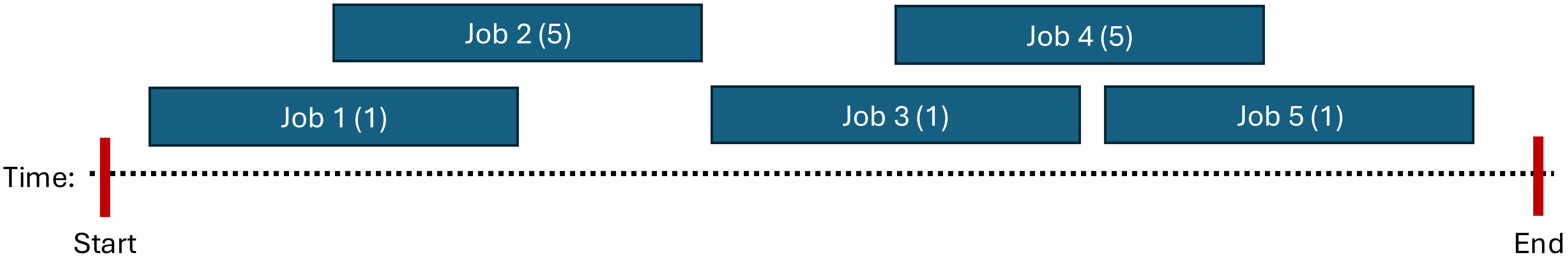
Weighted Case

- **Question:** Does this greedy algorithm work for other weights?



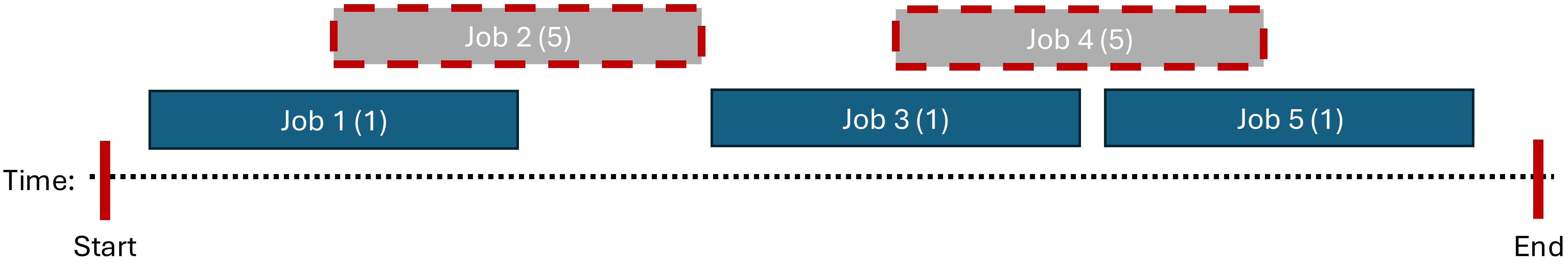
Weighted Case

- **Question**: Does this greedy algorithm work for other weights?
- **Answer**: No, there could be a situation where the number of jobs isn't the thing to maximize.



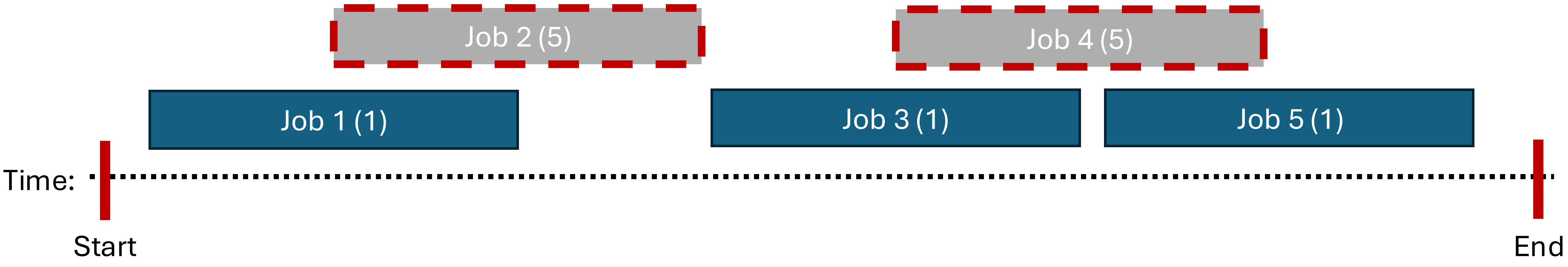
Weighted Case

- **Question:** Does this greedy algorithm work for other weights?
- **Answer:** No, there could be a situation where the number of jobs isn't the thing to maximize.



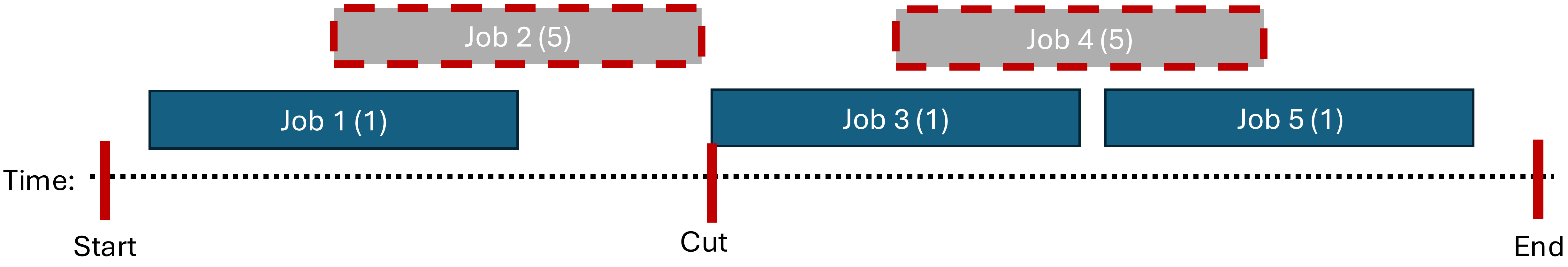
Divide & Conquer Approach

- **Question:** How would divide and conquer work for this problem?



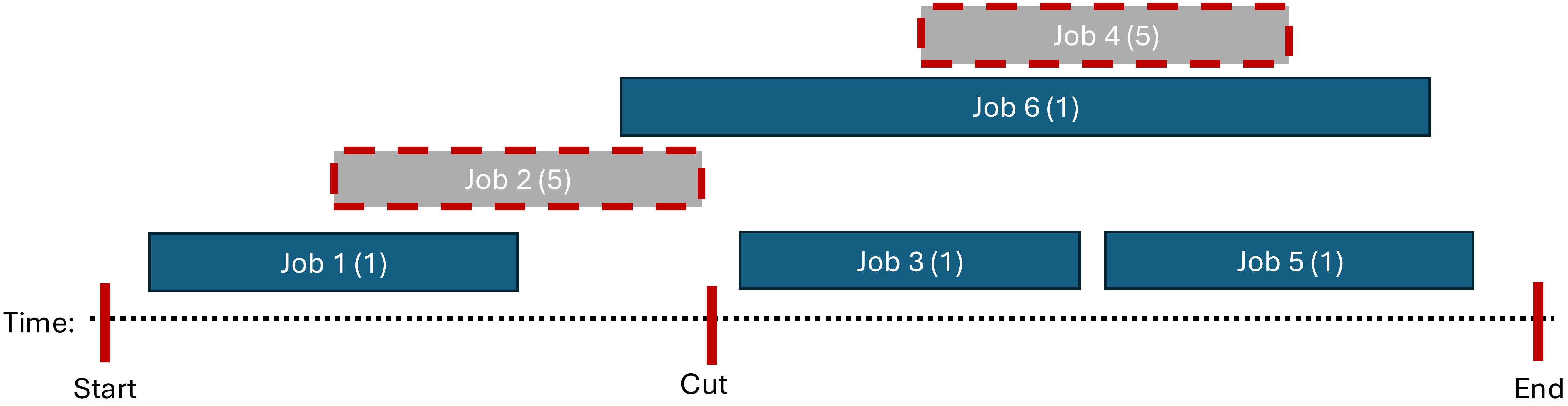
Divide & Conquer Approach

- **Question:** How would divide and conquer work for this problem?
- **Answer:** You might try to split the time in half or even put half of the jobs in both halves.



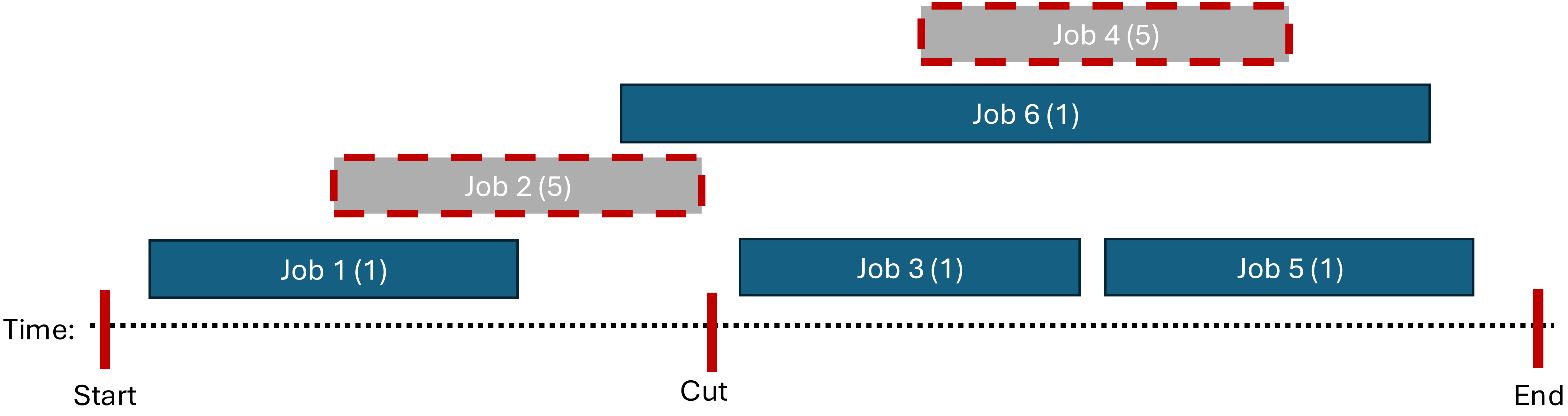
Divide & Conquer Approach

- **Question:** How would divide and conquer work for this problem?
- **Answer:** You might try to split the time in half or even put half of the jobs in both halves.
  - However, some problems may not split as easily…



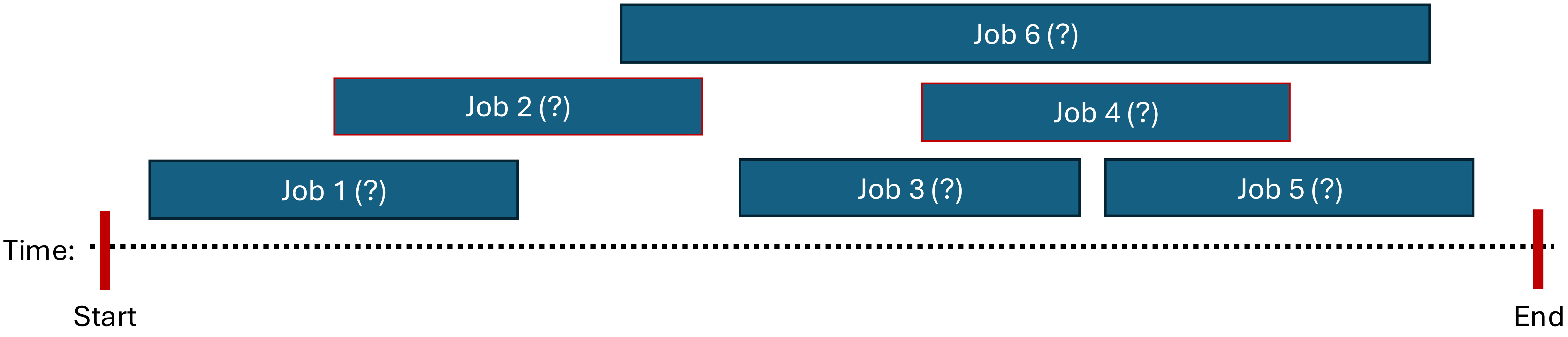
Divide & Conquer Approach

- **Question:** How would divide and conquer work for this problem?
- **Answer:** You might try to split the time in half or even put half of the jobs in both halves.
  - However, some problems may not split as easily...



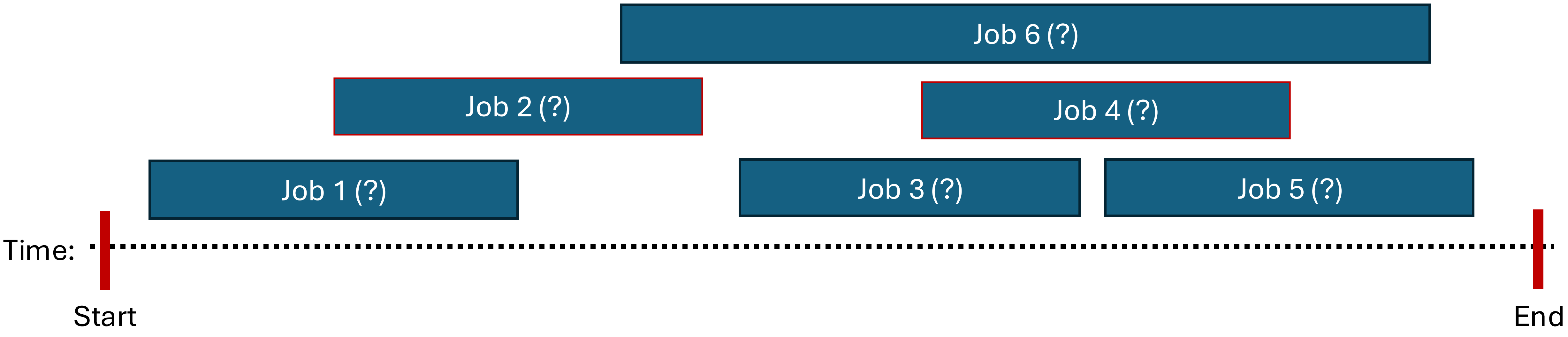
Divide & Conquer Approach

- **Question:** Consider an arbitrary instance with optimal solution OPT. What do we know about job 1?



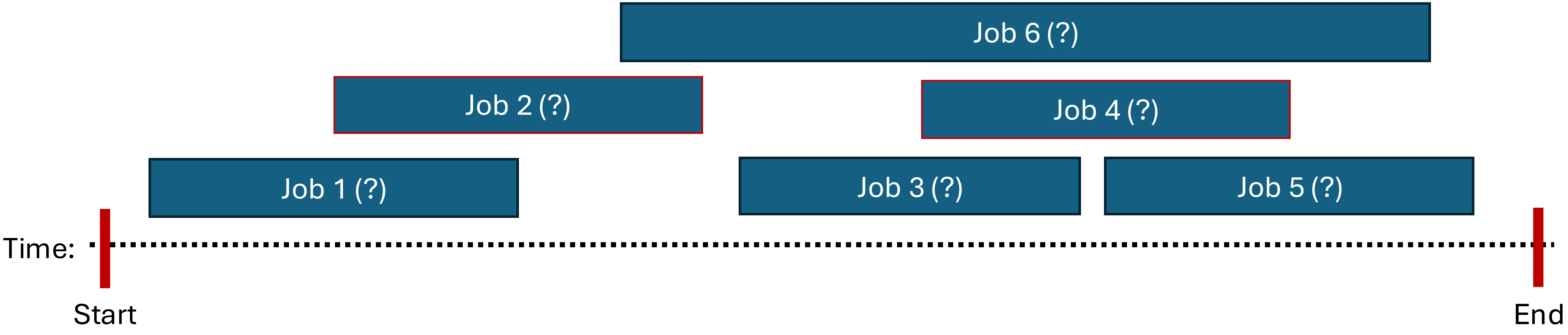
Binary Choice

- **Question:** Consider an arbitrary instance with optimal solution OPT. What do we know about job 1?
- **Answer:** It is either in OPT or it is not.



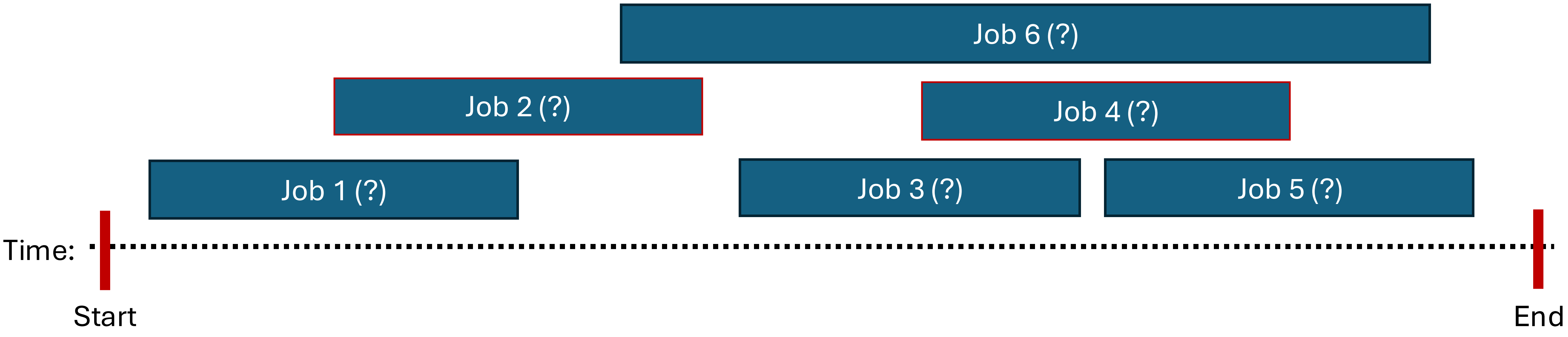
Binary Choice

- In a greedy algorithm we are assuming about if job 1 is in the optimal solution.
  - Our greedy rule doesn't work anymore if we don't know the weight.
- **Question**: Why not try both options?



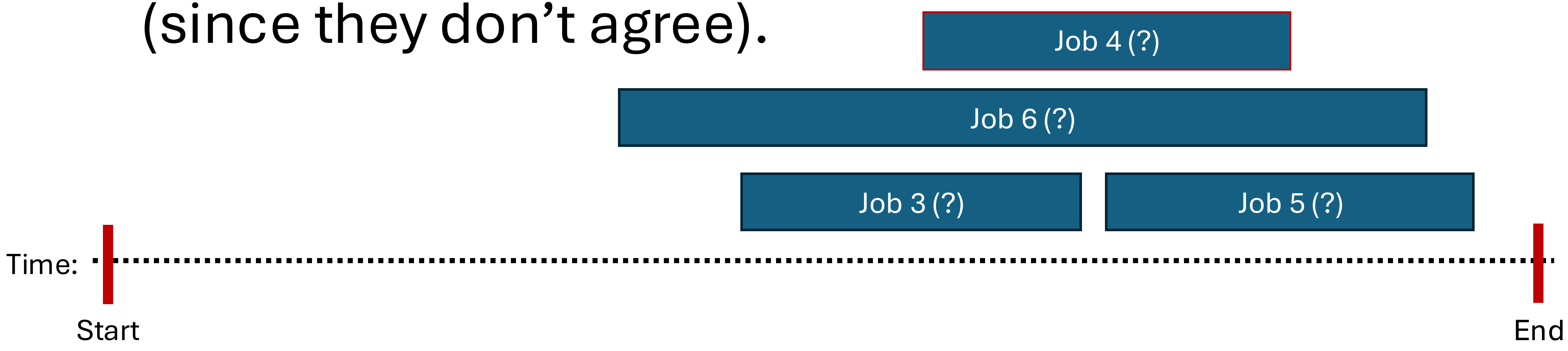
Binary Choice

- Suppose I gave you the hint that Job 1 was in OPT.
- **Question**: What do you need to do to find the rest of OPT?



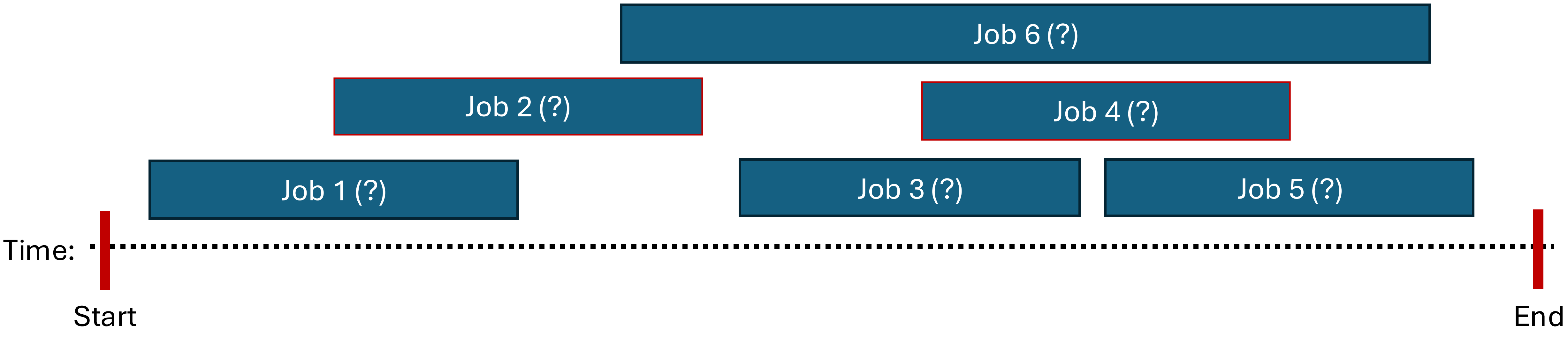
Binary Choice

- Suppose I gave you the hint that Job 1 was in OPT.
- **Question**: What do you need to do to find the rest of OPT?
- **Answer**: Recurse on a job without Job 1 or Job 2 (since they don't agree).



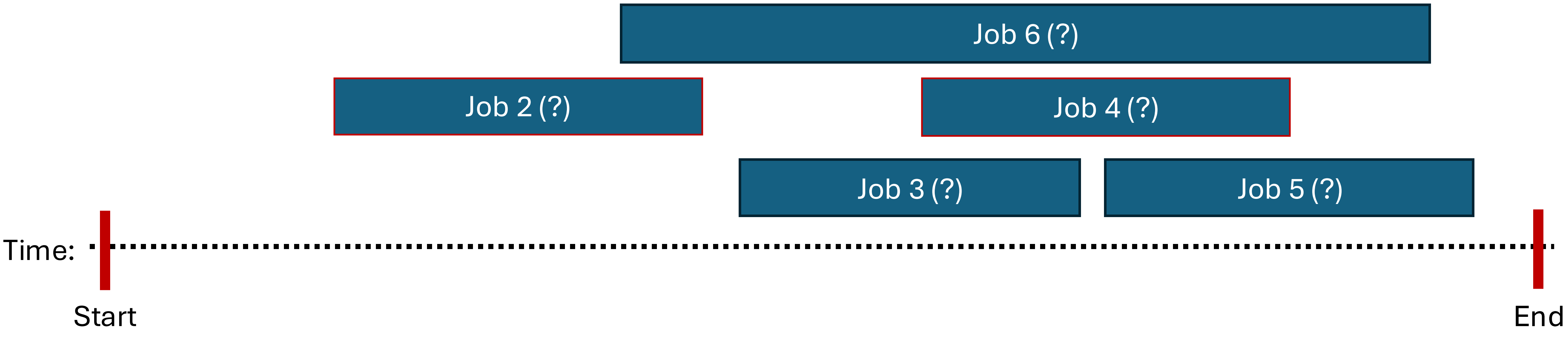
Binary Choice

- Suppose I gave you the hint that Job 1 was in OPT.
- **Question**: What do you need to do to find the rest of OPT?



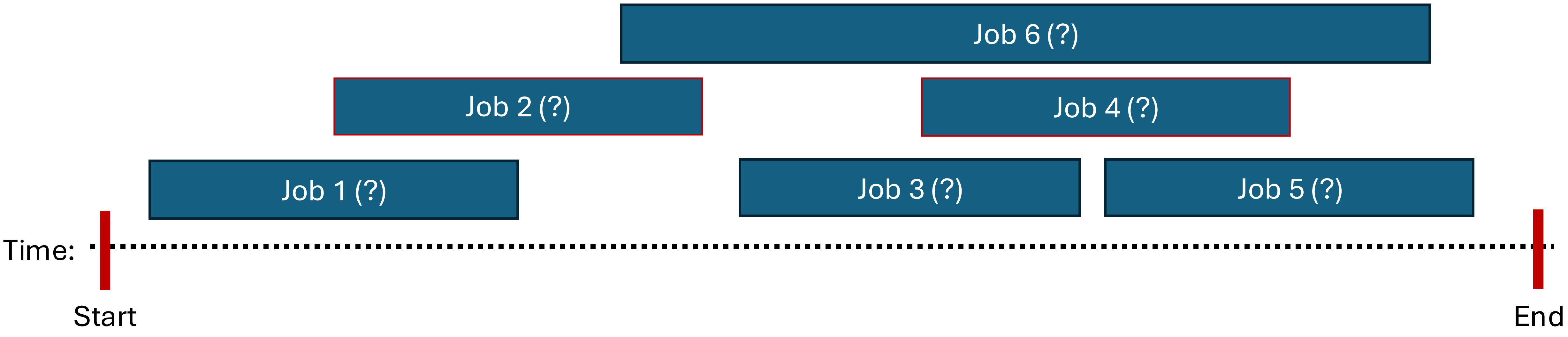
Binary Choice

- Suppose I gave you the hint that Job 1 was not in OPT.
- **Question**: What do you need to do to find the rest of OPT?
- **Answer**: Recurse on a job without Job 1.



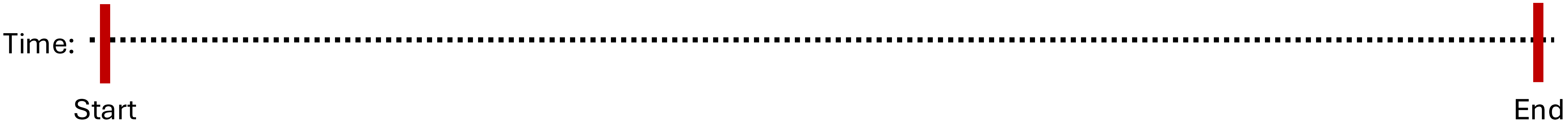
Binary Choice

- **Observation**: In both cases, we found a smaller instance of the problem to consider!



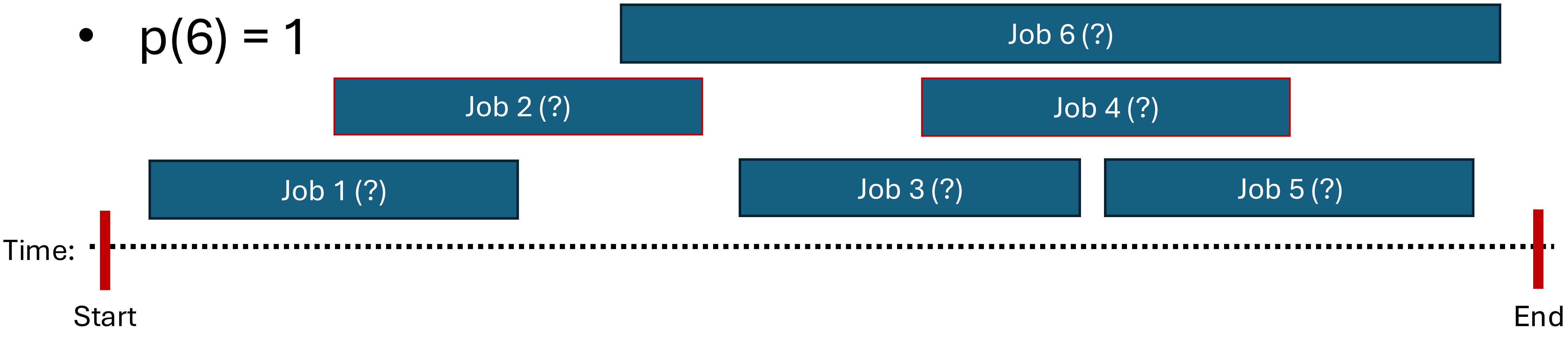
Binary Choice

- Assume our list of n jobs are sorted by finish times.
- For all j $\in [n]$,
  - Let S_j be the optimal solution on the first j jobs.
  - Let OPT(j) be the value of that solution.
  - Let p(j) be largest i such that i $<$ j and Job i is computable with Job j.
    - Let p(j) = 0 if no jobs exist.



Binary Choice

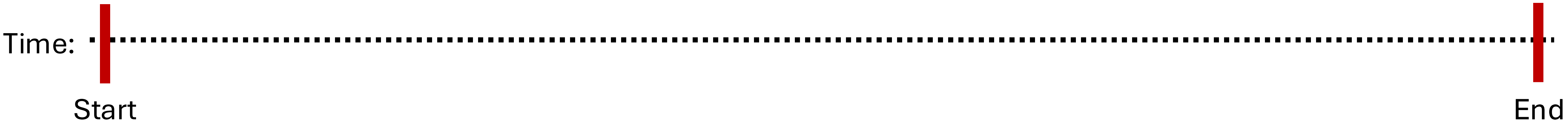
- p(1) = 0
- p(2) = 0
- p(3) = 2
- p(4) = 2
- p(5) = 3
- p(6) = 1



Binary Choice

- Now we can write

$$OPT(j) \;=\; max\{(v_j + OPT(p(j)), OPT(j-1)\}$$

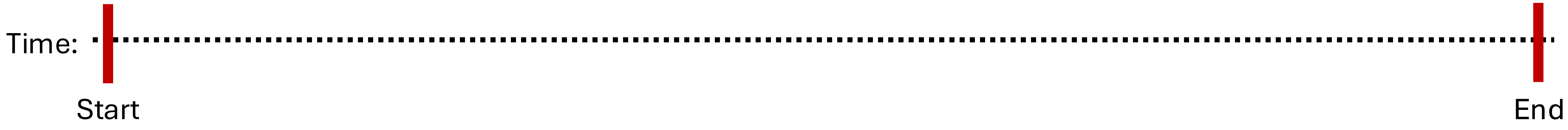


Binary Choice

- Now we can write

$$OPT(j) \; = \; max\{(v_j + OPT(p(j)), OPT(j-1)\}$$

English: The optimum solution for the first j jobs either uses Job j or it does not. The maximum of these two choices is the optimum.



Binary Choice

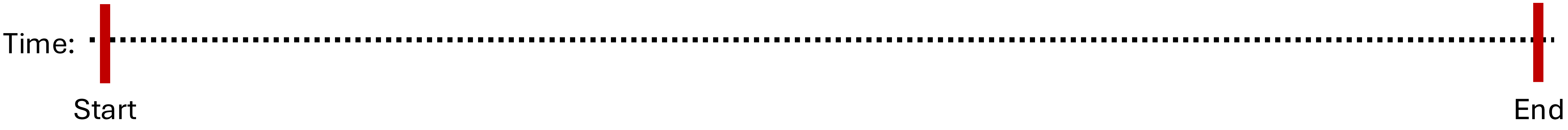
- Now we can write

We take Job J.

We consider the next job.

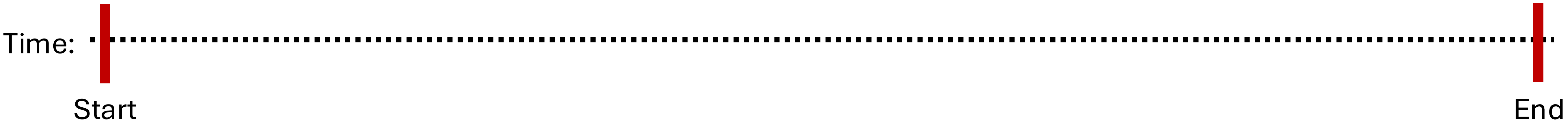
$$OPT(j) \;=\; max\{(v_j + OPT(p(j)), OPT(j-1)\}$$

English: The optimum solution for the first j jobs either uses Job j or it does not. The maximum of these two choices is the optimum.



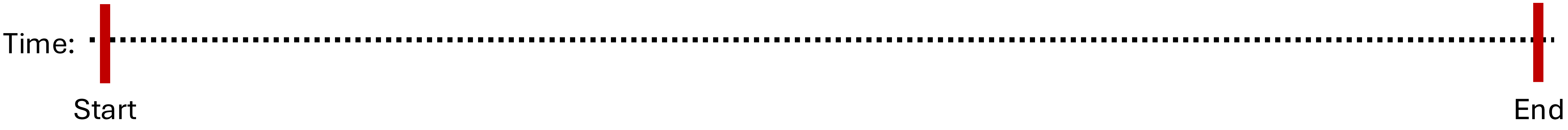
Binary Choice Algorithm

```
Brute-Force(L):
     Sort L by job finish times.
     Compute p[i] for each i using binary search.
     Return Compute-Opt(n)

Compute-Opt(j):
     If (j == 0):
           Return 0
     else:
          Return Max(Compute-Opt(j-1), v[j] + Compute-Opt(p[j])}
```



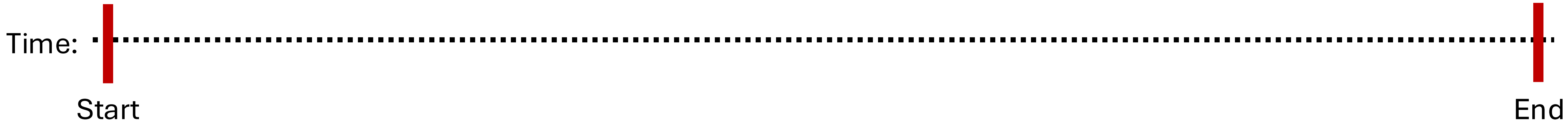
Question: What is the runtime?

```
Brute-Force(L):
     Sort L by job finish times.
     Compute p[i] for each i using binary search.
     Return Compute-Opt(n)

Compute-Opt(j):
     If (j == 0):
           Return 0
     else:
          Return Max(Compute-Opt(j-1), v[j] + Compute-Opt(p[j]))}
```



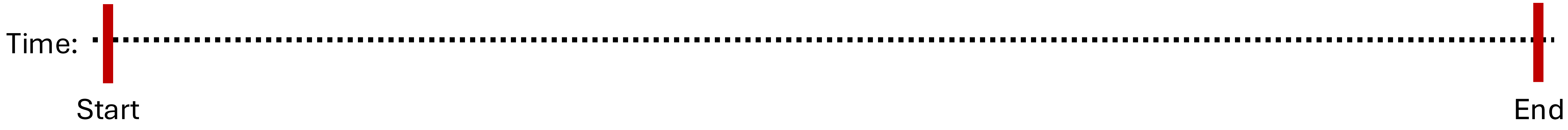
Question: What is the runtime?

```
Brute-Force(L):
     Sort L by job finish times.                           } O(nlog(n))
     Compute p[i] for each i using binary search.          }
     Return Compute-Opt(n)

Compute-Opt(j):
     If (j == 0):
           Return 0
     else:
          Return Max(Compute-Opt(j-1), v[j] + Compute-Opt(p[j])}
```



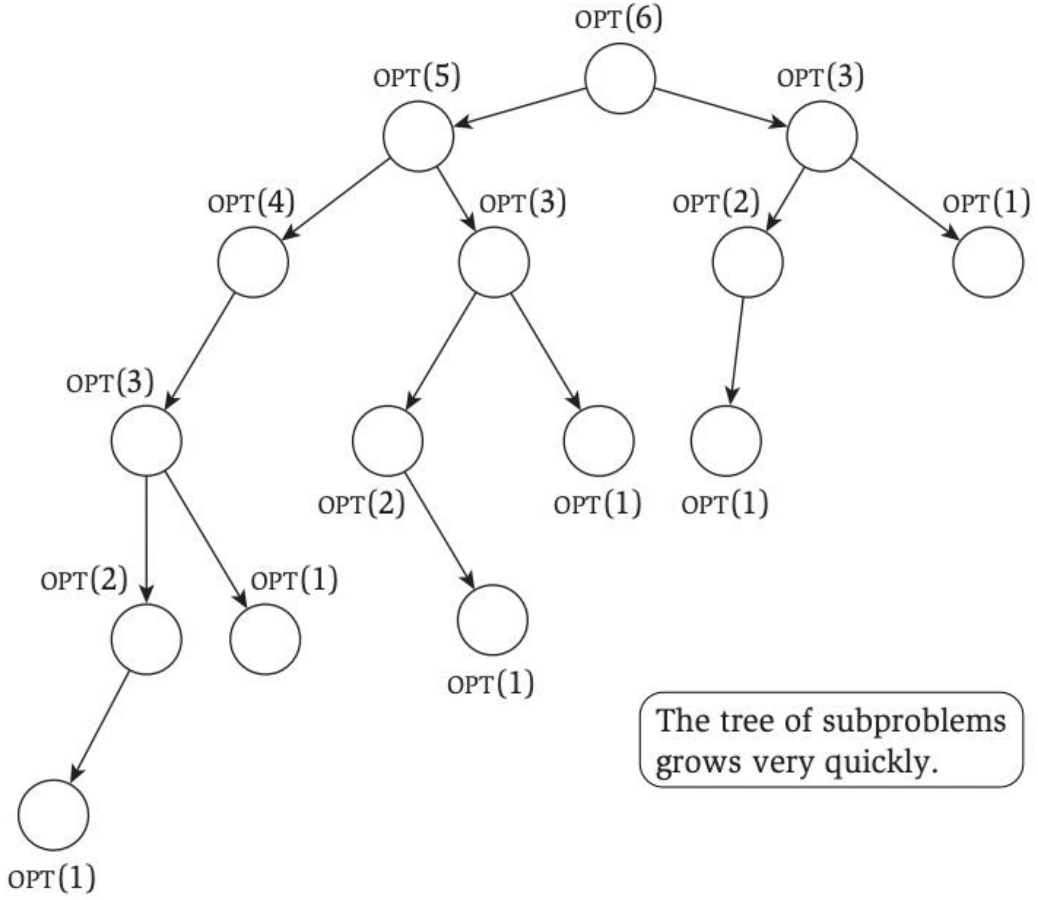
Answer: Could be exponential

```
Brute-Force(L):
     Sort L by job finish times.
     Compute p[i] for each i using binary search.
     Return Compute-Opt(n)

Compute-Opt(j):
     If (j == 0):
           Return 0
     else:
          Return Max(Compute-Opt(j-1), v[j] + Compute-Opt(p[j])}
```



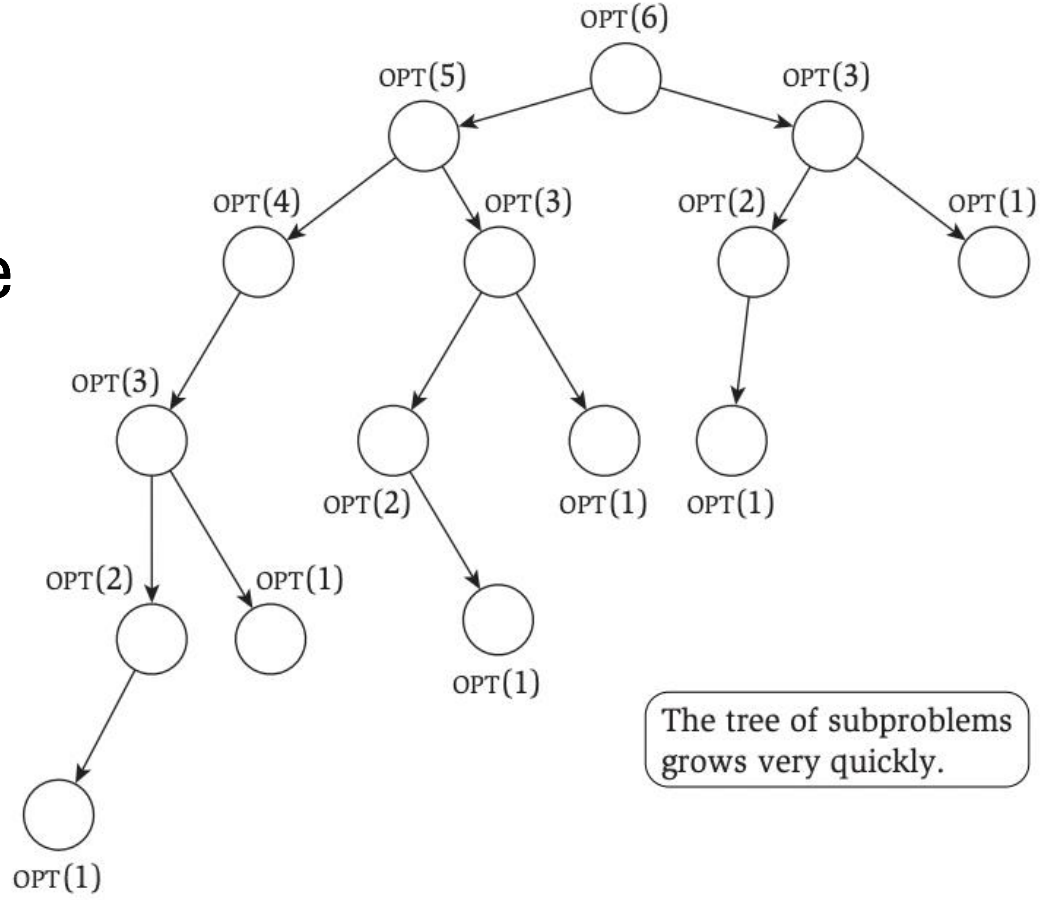
Recursion Tree

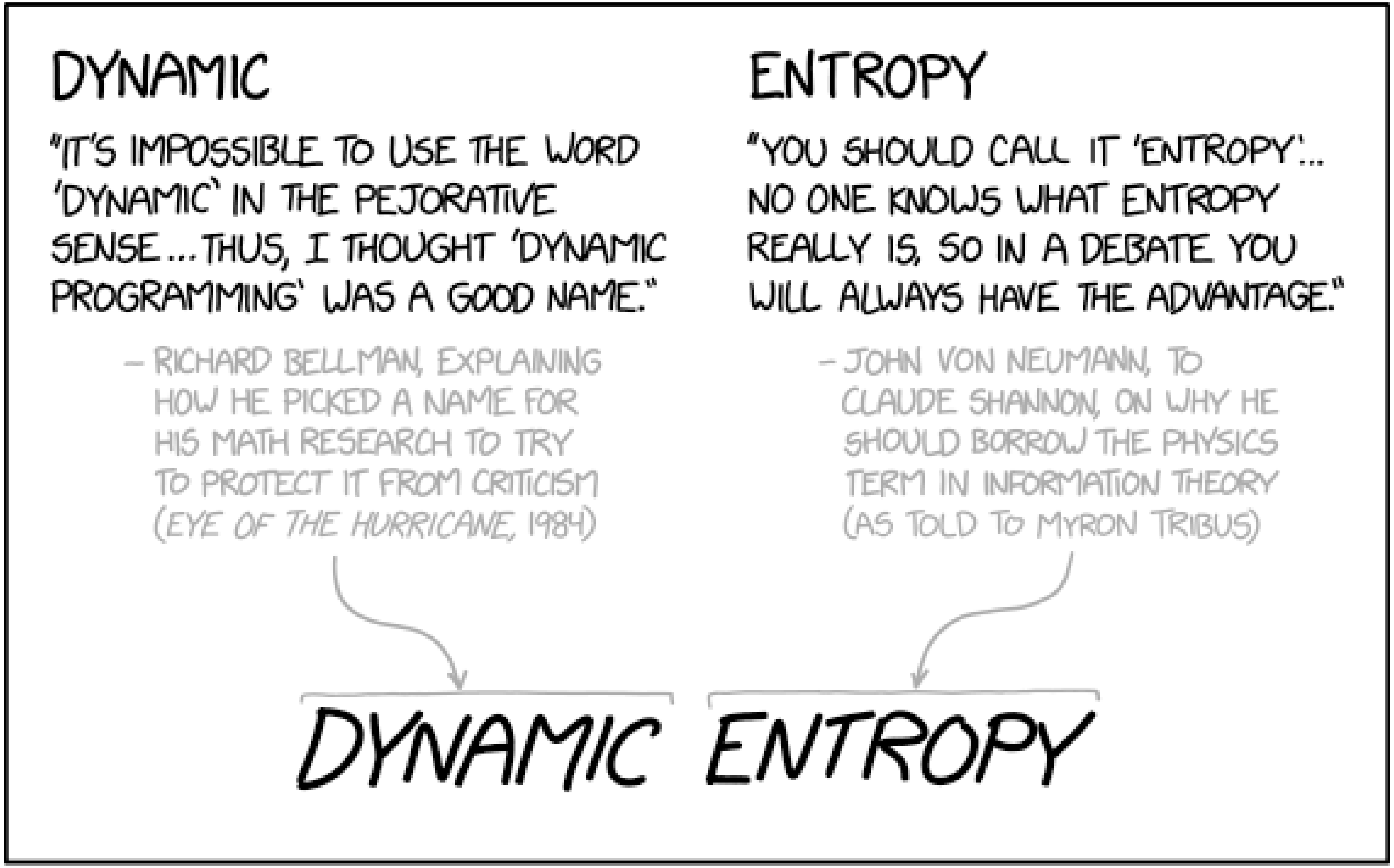
- For each OPT(i) we draw an arrow to the subproblems we need to solve to solve it.
- It is possible that the tree has linear depth, and each internal node has two children.
- Notice that some problems appear more than once!



Memoization

- Notice that some problems appear more than once!
- What if our algorithm never computed the answer to the same subproblem more than once?
- Let's keep track of our answers using an array.



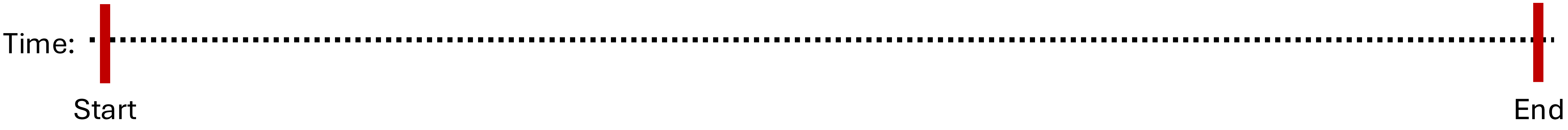


SCIENCE TIP: IF YOU HAVE A COOL CONCEPT YOU NEED A NAME FOR, TRY "DYNAMIC ENTROPY."

https://xkcd.com/2318/

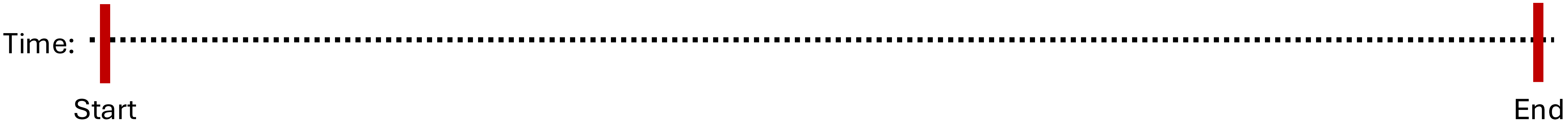
Remember Remember

```
Brute-Force(L):
     Sort L by job finish times.
     Compute p[i] for each i using binary search.
     Global M = [], M[0] = 0
     Return Compute-Opt(n)

M-Compute-Opt(j):
     If j not in M:
          M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]
     Return M[j]
```



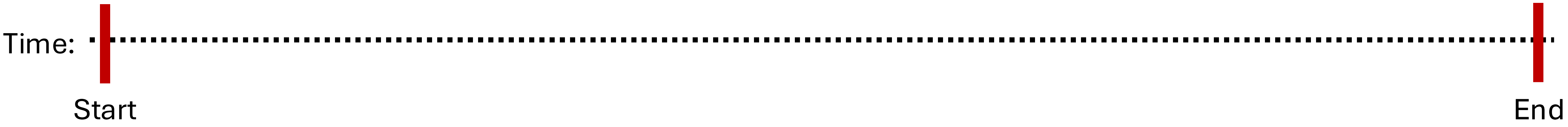
Memoization Runtime

```
Brute-Force(L):
     Sort L by job finish times.
     Compute p[i] for each i using binary search.
     Global M = [], M[0] = 0
     Return Compute-Opt(n)

M-Compute-Opt(j):
     If j not in M:
          M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]
     Return M[j]
```



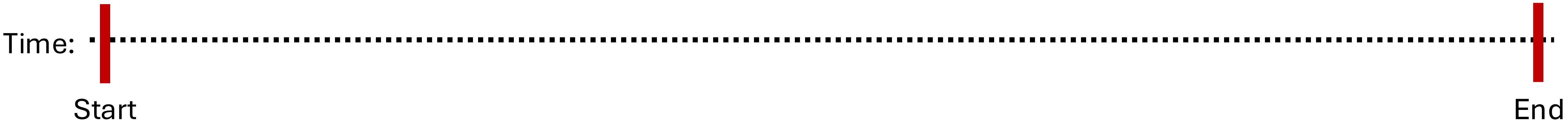
Memoization Runtime

- We will show that the runtime is O(nlog(n)).
  - The preprocessing takes O(nlog(n)) time.
  - The M-Compute-Opt(n) call takes O(n) time.



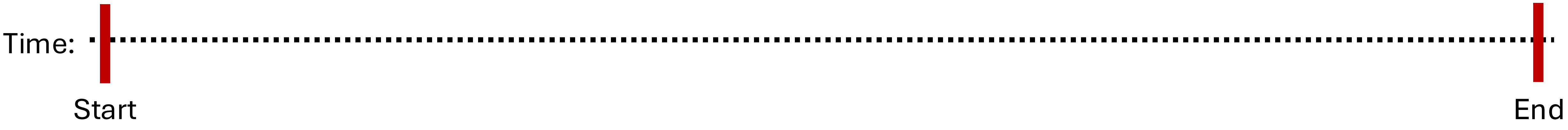
Memoization Runtime

- The M-Compute-Opt(n) call takes O(n) time.
  - To bound the runtime, we will introduce a ”progress measure”. Namely, we will track how many entries in M are uninitialized.
  - Each time we initialize an entry of M, we make two recursive calls which takes constant time.
  - Since M will only have at most O(n) entries, it follows that the runtime is at most O(n) as desired.

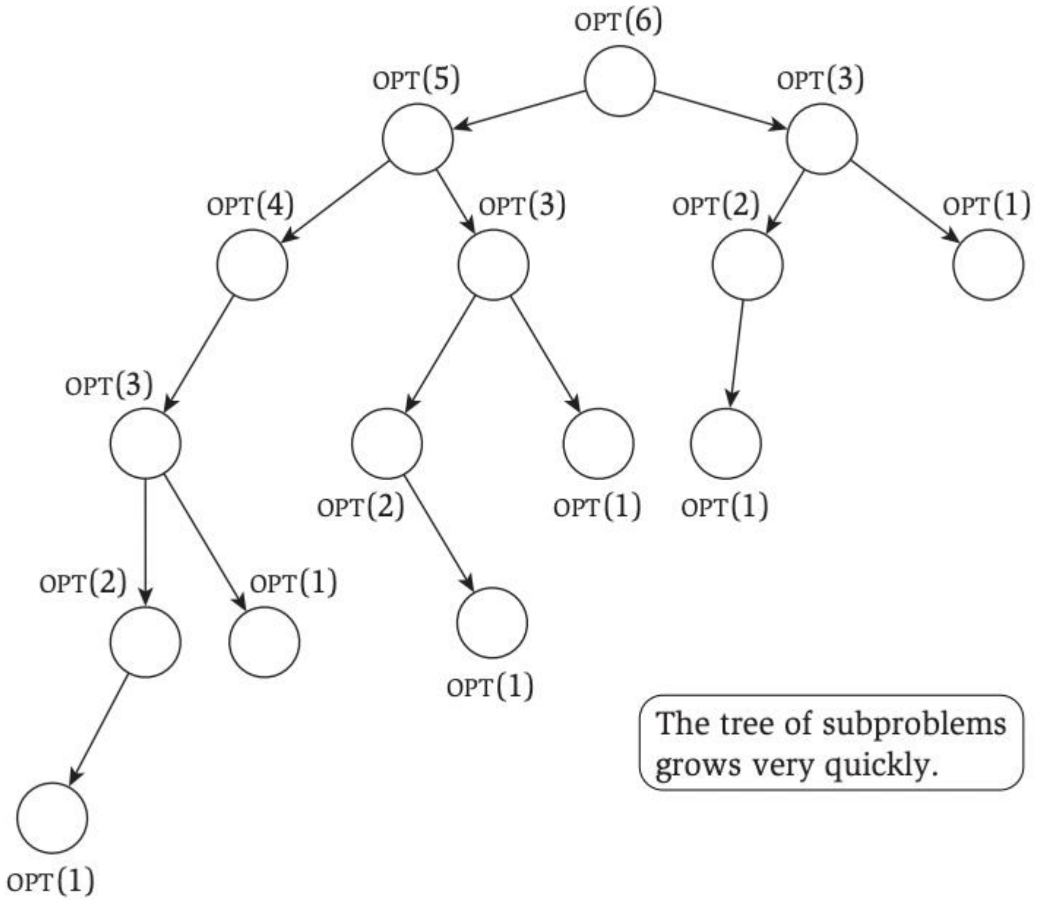


Top-Down Dynamic Programming

```
Brute-Force(L):
     Sort L by job finish times.
     Compute p[i] for each i using binary search.
     Global M = [], M[0] = 0
     Return Compute-Opt(n)

M-Compute-Opt(j):
     If j not in M:
          M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]
     Return M[j]
```



Top-Down Dynamic Programming



Next Time

- Recover Optimal Solutions
- Bottom-Up Dynamic Programming
- Process of coming up with Dynamic Programming algorithm.