Lecture 11: Dynamic Programming 1

CS 341: Algorithms

Tuesday, Feb 12th 2019

Outline For Today

1. Linear Independent Set
2. Recipe of a Dynamic Programming Algorithm
3. Weighted Activity Selection

Outline For Today

1. Linear Independent Set
2. Recipe of a Dynamic Programming Algorithm
3. Weighted Activity Selection

Linear Independent Set

- Input: Line graph G(V, E), and *weights $w_v \geq 0$ on each vertex*
- Output: The *max-weight independent set* of vertices in G;

  i.e. mutually non-adjacent set of vertices with max-weight

Linear Independent Set

- Input: Line graph G(V, E), and *weights w_v on each vertex*
- Output: The *max-weight independent set* of vertices in G;

  i.e. mutually non-adjacent set of vertices with max-weight

Independent Set 1: {A, D}
Weight: 4

Linear Independent Set

- Input: Line graph G(V, E), and *weights w_v on each vertex*
- Output: The *max-weight independent set* of vertices in G;

  i.e. mutually non-adjacent set of vertices with max-weight

Independent Set 2: {A, C}
Weight: 7

Linear Independent Set (IS)

- Input: Line graph G(V, E), and *weights w_v on each vertex*
- Output: The *max-weight independent set* of vertices in G;

  i.e. mutually non-adjacent set of vertices with max-weight

***Max Independent Set**: {B, D}*
Weight: 8

Possible Approaches (1)

- Brute-force Search: exponential number of different independent sets

Possible Approaches (2)

- Greedy:

  Let S be ∅

  1. while (cannot pick any vertices)
  2. let v be max-weight that is not adjacent to vertices in S
  3. add v to S

Possible Approaches (2)

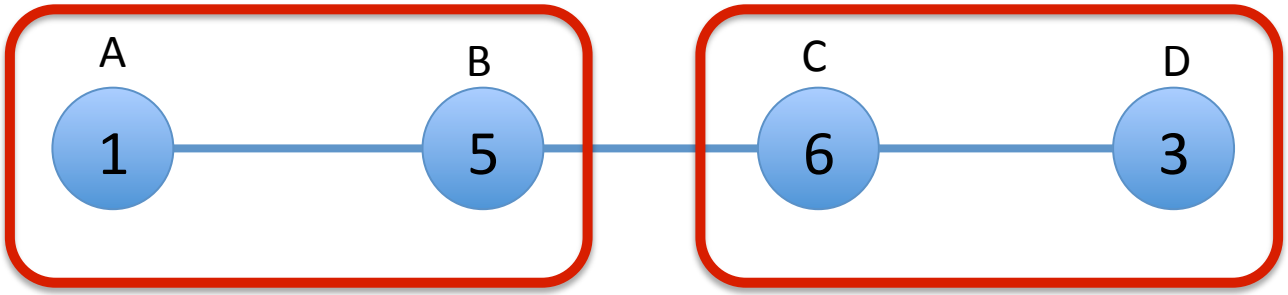
- Greedy: Let S be ∅

  1. while (cannot pick any vertices)
  2. let v be max-weight that is not adjacent to vertices in S
  3. add v to S

Greedy Solution: {A, C}
Weight: 7
Not Correct!

Possible Approaches (3)

- Divide and Conquer
  1. Divide into L, R
  2. Find max IS on L and R
  3. Merge ISs of L and R



Possible Approaches (3)

- Divide and Conquer
  1. Divide into L, R
  2. Find max IS on L and R
  3. Merge ISs on L and R

Can be conflicts at the boundary

DC approach can be made to work but will be slow.

A New Algorithm: First Steps

- Reason about what the optimal solution looks like

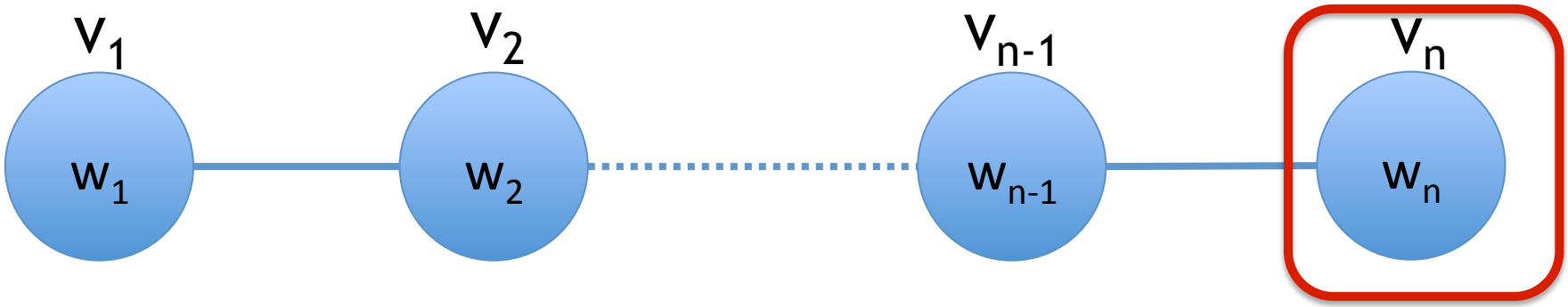
in terms of optimal solutions to sub-problems

Let S be a max-weight IS*

Consider v_n

A Claim That Doesn't Require a Proof:

There are 2 possible cases: (1) $V_n \notin S^*$ or (2): $v_n \in S^*$



Case 1: $V_n \notin S^*$

Consider $G` = G - v_n$

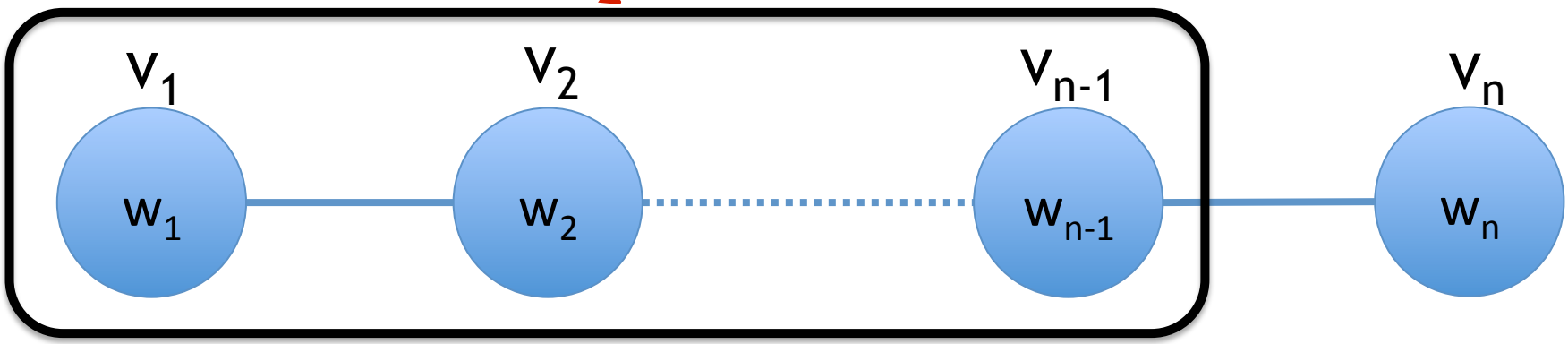
Q: What can we assert about S^* in $G`$?

A: S^* is max-weight IS in $G`$!

Proof: Assume $\exists S`$ in $G`$ s.t. $w(S`) > w(S^*)$ => since $S`$ is a valid IS in G, S^* cannot be the max in G.

S^* is optimal for subproblem $G`$

$G`$



Case 2: $V_n \in S^*$

Q1: What can we assert about v_{n-1}?

A1: $v_{n-1} \notin S^*$! (would violate independence of S^*)

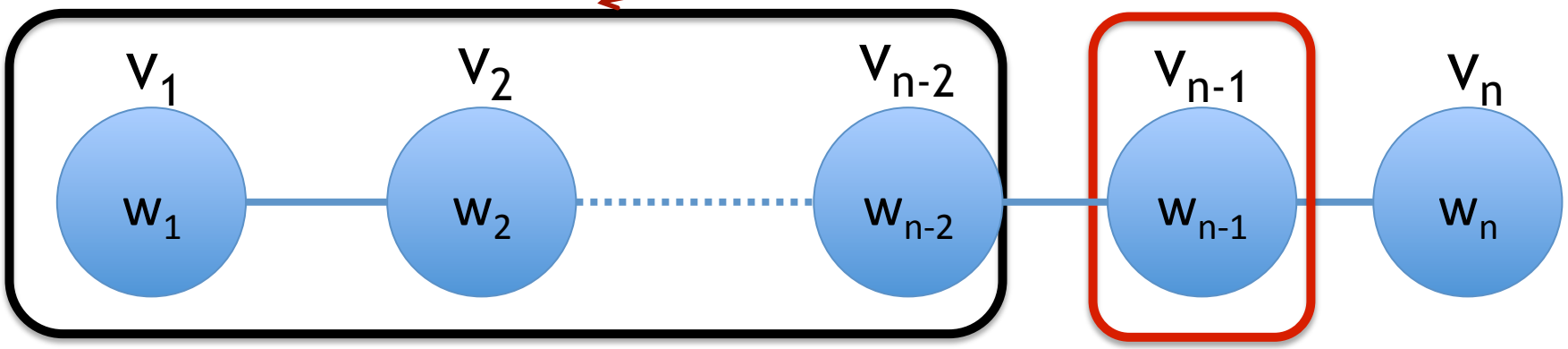
Let G`` be G – $\{v_n, v_{n-1}\}$

Q2: What can we assert about $S^*-\{v_n\}$ in G``?

A2: $S^*-\{v_n\}$ is optimal in G``.

G``

Claim: $S^*-\{v_n\}$ is optimal for subproblem G``



Proof of $S^*-\{v_n\}$'s optimality in G``

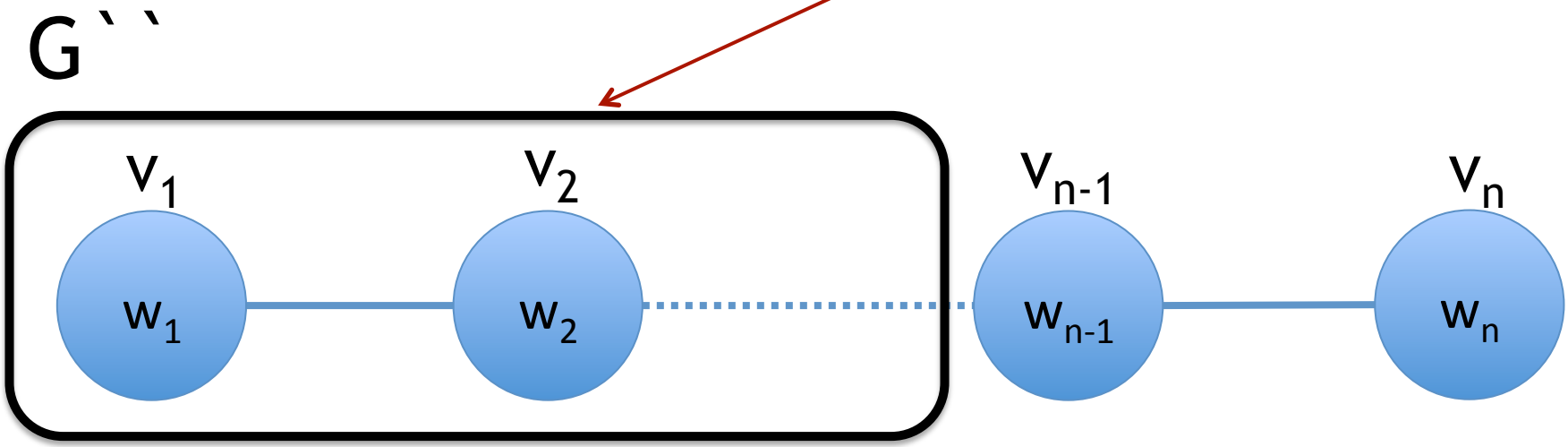
Claim: $S^*-\{v_n\}$ is an IS in G``. Assume it's not optimal.

Let S`` be an IS for G``with $w(S``) > w(S^* - \{v_n\})$

Then note $S`` + \{v_n\}$ is an IS for G, and has weight

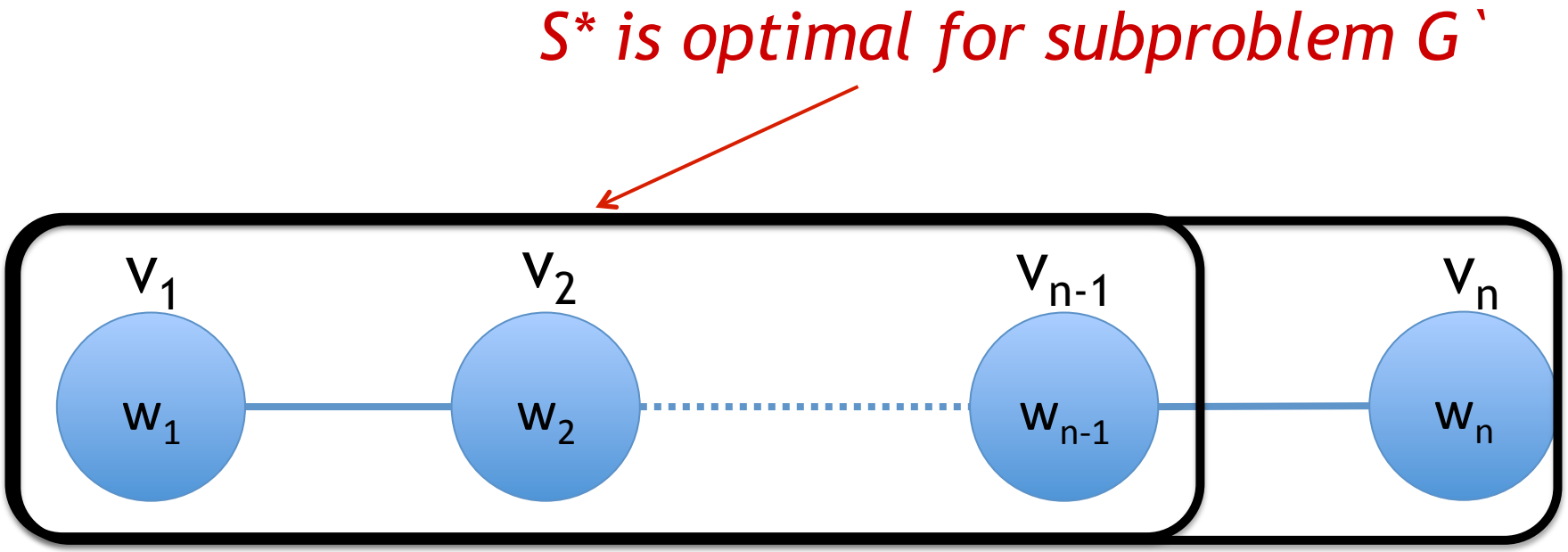
> S, contradicting S*'s optimality.*

$S^*-\{v_n\}$ is optimal for subproblem G``



Summary of 2 Cases

1. $v_n \notin S^*$ => S^* is optimal for $G` = G - \{v_n\}$

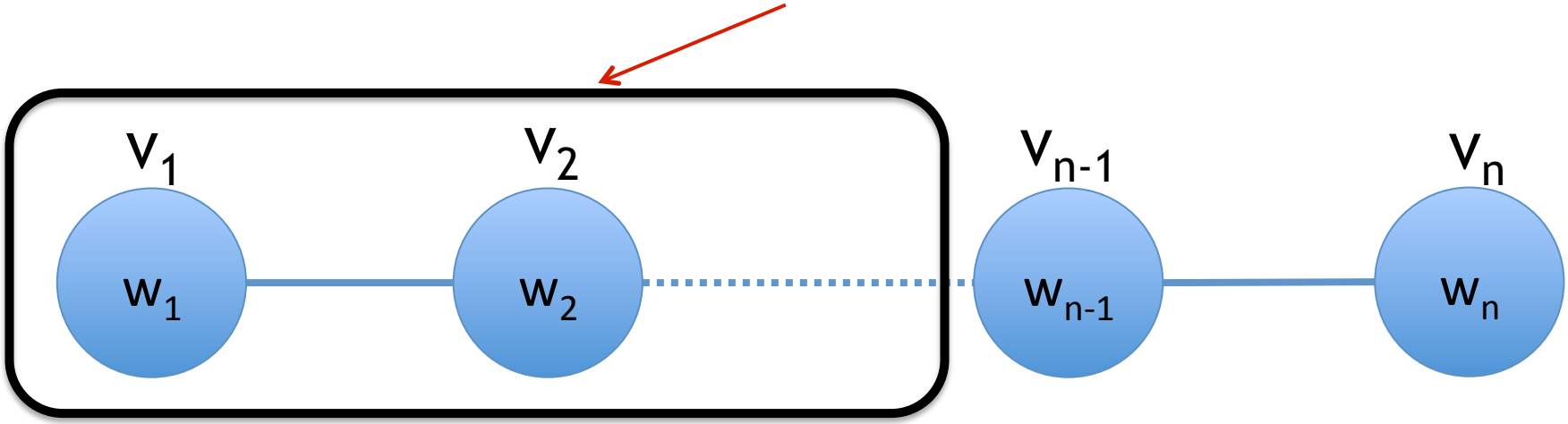


Summary of 2 Cases

1. $v_n \notin S^* \Rightarrow S^*$ is optimal for $G` = G-\{v_n\}$
2. $v_n \in S^* \Rightarrow S^* - \{v_n\}$ is optimal for $G`` = G-\{v_n, v_{n-1}\}$

If we knew which case we're in, we'd know how to recurse, and be done!

$S^*-\{v_n\}$ is optimal for subproblem $G``$



A Possible Recursive Algorithm

Recurse on both cases and return the better solution.

Recursive-Linear-IS-1: (G(V, E) and weights)

1. Let S_1 = Recursive-Linear-IS-1(G`)
2. Let S_2 = Recursive-Linear-IS-1(G``)
3. return the better of S_1 or $S_2 \cup \{v_n\}$

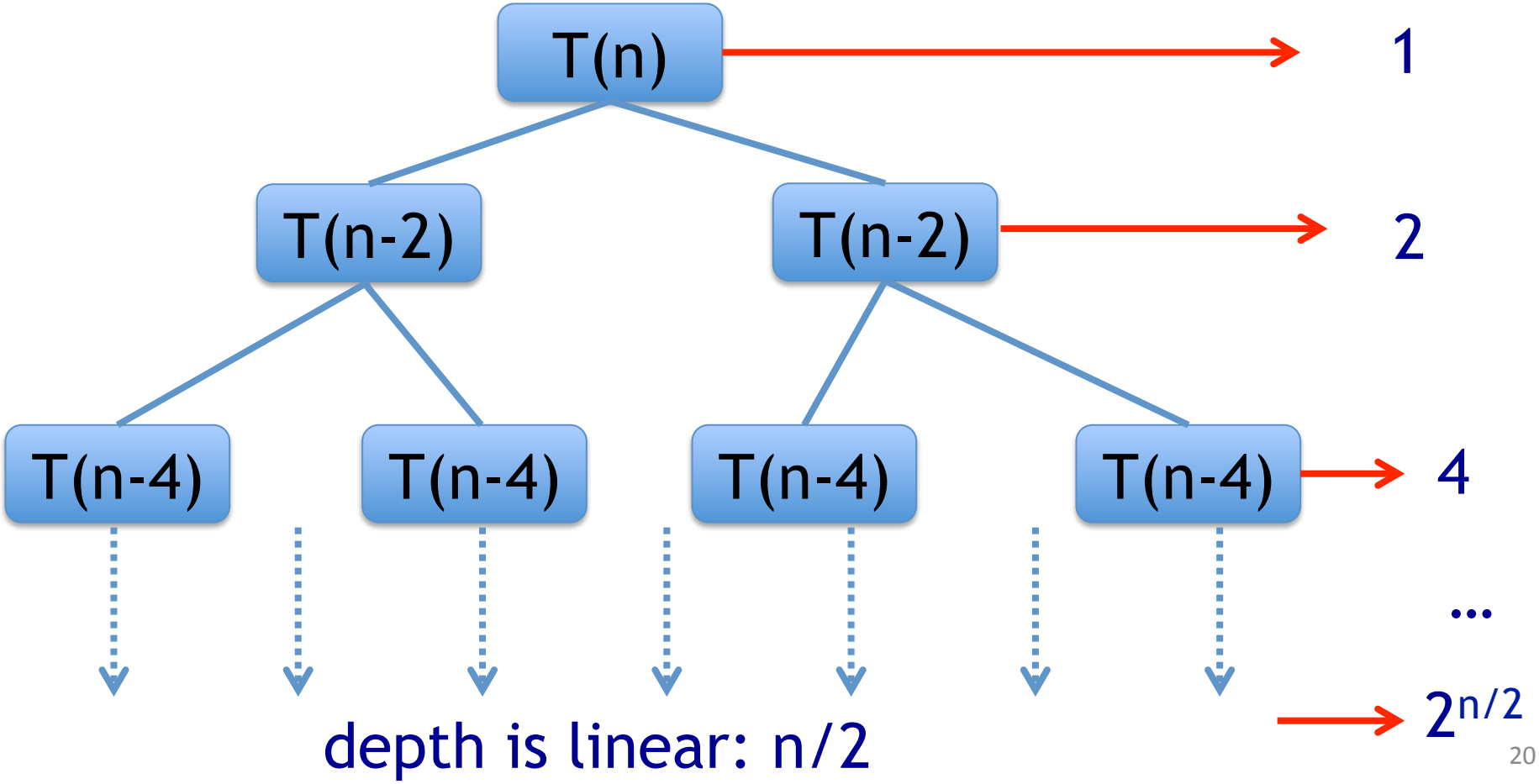
Good news: The algorithm is correct!

Problem: This looks like brute-force search!

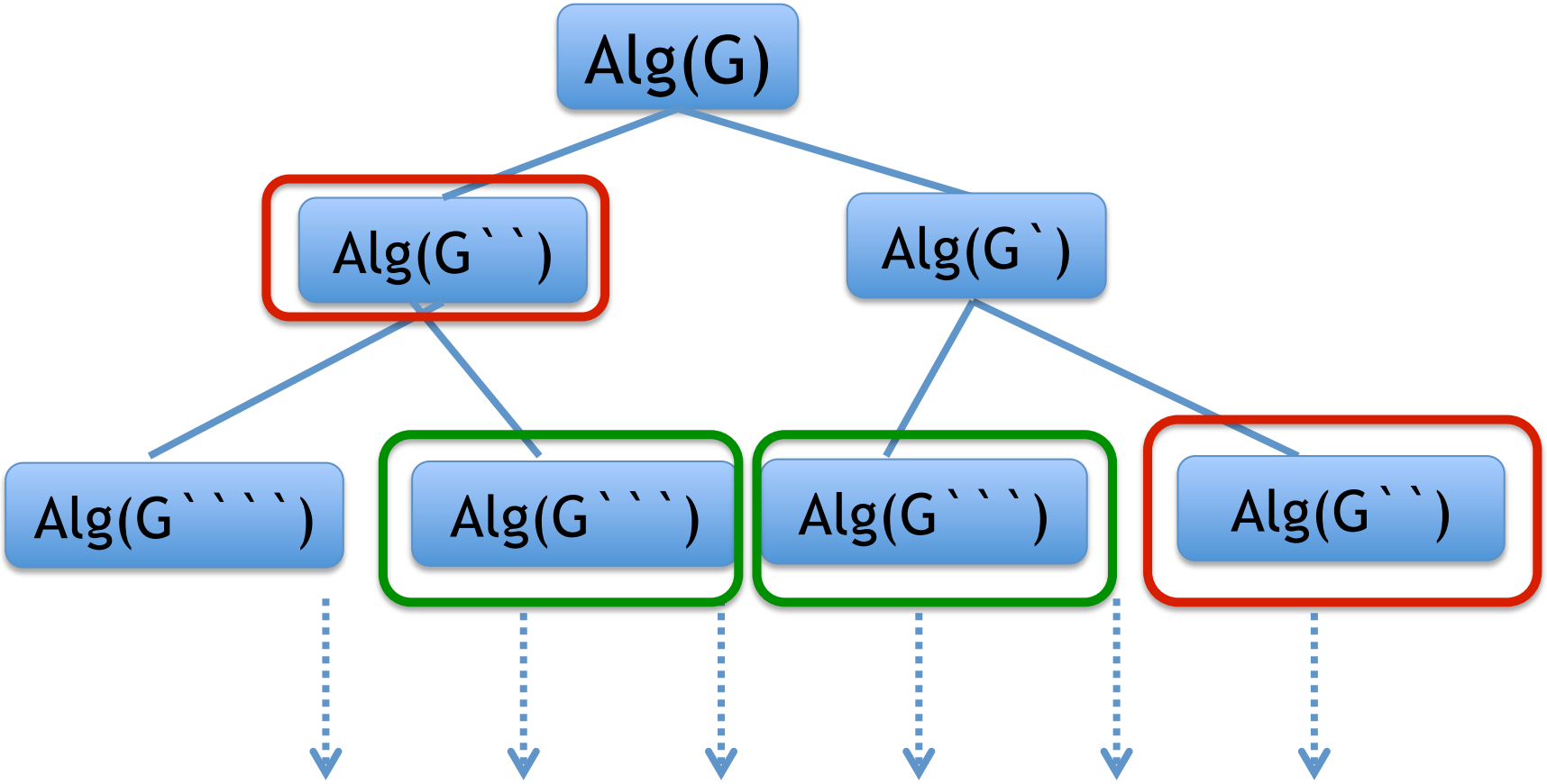
Why is The Runtime Exponential?

$T(n) = T(n-1) + T(n-2) + O(1)$

$T(n) \geq 2T(n-2) + O(1) = \Omega(2^n)$



Q: How many distinct recursive calls?



Answer: only n

b/c each input is a prefix of $v_1 \ldots v_n$.

The exponential time is due to large redundancy

Fix 1: Memoization

Store solutions to subproblems each time they're solved. Afterwards, just lookup the solution, instead of computing it.

Runtime: O(n) (Exercise: We store each subproblem once => O(n) stores. Need to count how many times you do lookups.)

Fix 2: Bottom-up Iterative Reformulation

Let G_i be the first prefix i vertices from left in G

Let A be an array of size n.

A[i] = max weight IS to G_i

```
procedure DP-Linear-IS(G(V,E)):
 Base Cases: A[0] =
```

Fix 2: Bottom-up Iterative Reformulation

Let G_i be the first prefix i vertices from left in G

Let A be an array of size n.

A[i] = max weight IS to G_i

```
procedure DP-Linear-IS(G(V,E)):
 Base Cases: A[0] = 0
             A[1] =
```

Fix 2: Bottom-up Iterative Reformulation

Let G_i be the first prefix i vertices from left in G

Let A be an array of size n.

A[i] = max weight IS to G_i

```
procedure DP-Linear-IS(G(V,E)):
 Base Cases: A[0] = 0
             A[1] = w1
  for i = 2,3,…,n:
    A[i] =
```

Fix 2: Bottom-up Iterative Reformulation

Let G_i be the first prefix i vertices from left in G

Let A be an array of size n.

A[i] = max weight IS to G_i

```
procedure DP-Linear-IS(G(V,E)):
 Base Cases: A[0] = 0
             A[1] = w_1
 for i = 2,3,…,n:
    A[i] = max{A[i-1], A[i-2] + w_i}
 return A[n]
```

Runtime & Correctness of DP-Linear-IS

Runtime: O(n) => only looping through the array.

Correctness: by induction (exercise)

Space: O(n) => but can do constant space (why?)

unless want to reconstruct the actual IS.

What if we also want to reconstruct the IS!

```
procedure DP-Linear-IS(G(V,E)):
 Base Cases: A[0] = 0
             A[1] = w_1
 for i = 2,3,…,n:
    A[i] = max{A[i-1], A[i-2] + w_i}
 return A[n]
```

Reconstructing the Optimal IS (1)

Option 1: Store for each A[i] also the IS_i.

=> quadratic space (should avoid in practice)

Option 2: Backward trace A and reconstruct the solution.

Claim: $v_i \in$ opt IS for G_i iff

$$w_i + w(\text{opt IS of } G_{i-2}) \geq w(\text{opt IS of } G_{i-1})$$

Proof: Same as S*'s optimality in G` and S*-{v_n}'s optimality in G`` (slides 13 & 15)

Q: Given this claim, how can we reconstruct the opt IS?

Reconstructing the Optimal IS (2)

Claim: $v_i \in$ opt IS for G_i iff

$$w_i + w(\text{opt IS of } G_{i-2}) \geq w(\text{opt IS of } G_{i-1})$$

A	1	3	7	…	83	85	90	90
Weights	1	3	6	…	2	4	7	3

Q: Is v_n in the optimal set?

A: No. w_n=3 + A[n-2]=85 ≤ A[n-1]=90 => We were in Case 1

Q: Is v_{n-1} in the optimal set?

A: Yes. w_{n-1}=7 + A[n-3]=83 ≥ A[n-2]=85 => We were in Case 2

Reconstructing the Optimal IS (3)

```
procedure DP-Linear-IS-Reconstruct(G(V,E)):
  A = DP-Linear-IS(G(V, E))
  let S = ∅
  i = n
  while i ≥ 0:
     if w_i + A[i-2] ≥ A[i-1]:
        put v_i into S; i = i - 2
     else: i = i - 1;
  return S
```

A	1	3	7	…	83	85	90	90
Weights	1	3	6	…	2	4	7	3

Outline For Today

1. Linear Independent Set
2. Recipe of a Dynamic Programming Algorithm
3. Weighted Activity Selection

Recipe of a DP Algorithm

1: Identify small # subproblems, (e.g., find opt IS in G_i: only n subproblems). We'll represent solutions to subproblems inside a one or multi-dimensional array (e.g. A).

2: quickly + correctly solve "larger" subproblems given solutions to smaller ones, usually via a recursive formula (e.g., A[i] = max $\{w_i + A[i-2], A[i-1]\}$)

3: After solving all subproblems, quickly compute final solution (e.g., return A[n])

How to Recognize a DP Solution

Step 1: Have a first-cut brute-force-like recursive algorithm, which expresses the larger solution in terms of solutions to smaller subproblems.

Step 2: Recognize that different branches have a lot of overlapping work.

Step 3: Recognize that there aren't actually that many different subproblems.

Q: What Does Dynamic Programming Mean?

Answer: Not much!

Richard Bellman: An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

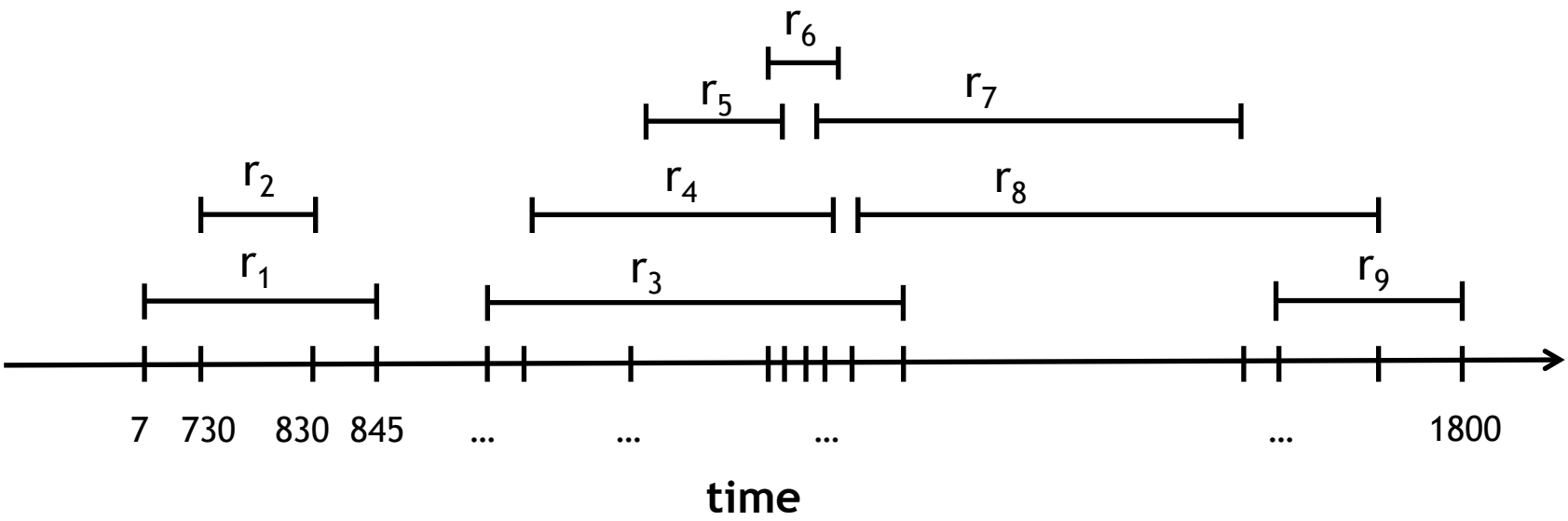
Outline For Today

1. Linear Independent Set
2. Recipe of a Dynamic Programming Algorithm
3. Weighted Activity Selection

Weighted Activity Selection (I.e. w/ Values)

- Input: 1 resource (lecture room) & n requests (e.g. events)

  where r_i has a start time s(i), finish time f(i), and *value* v_i.

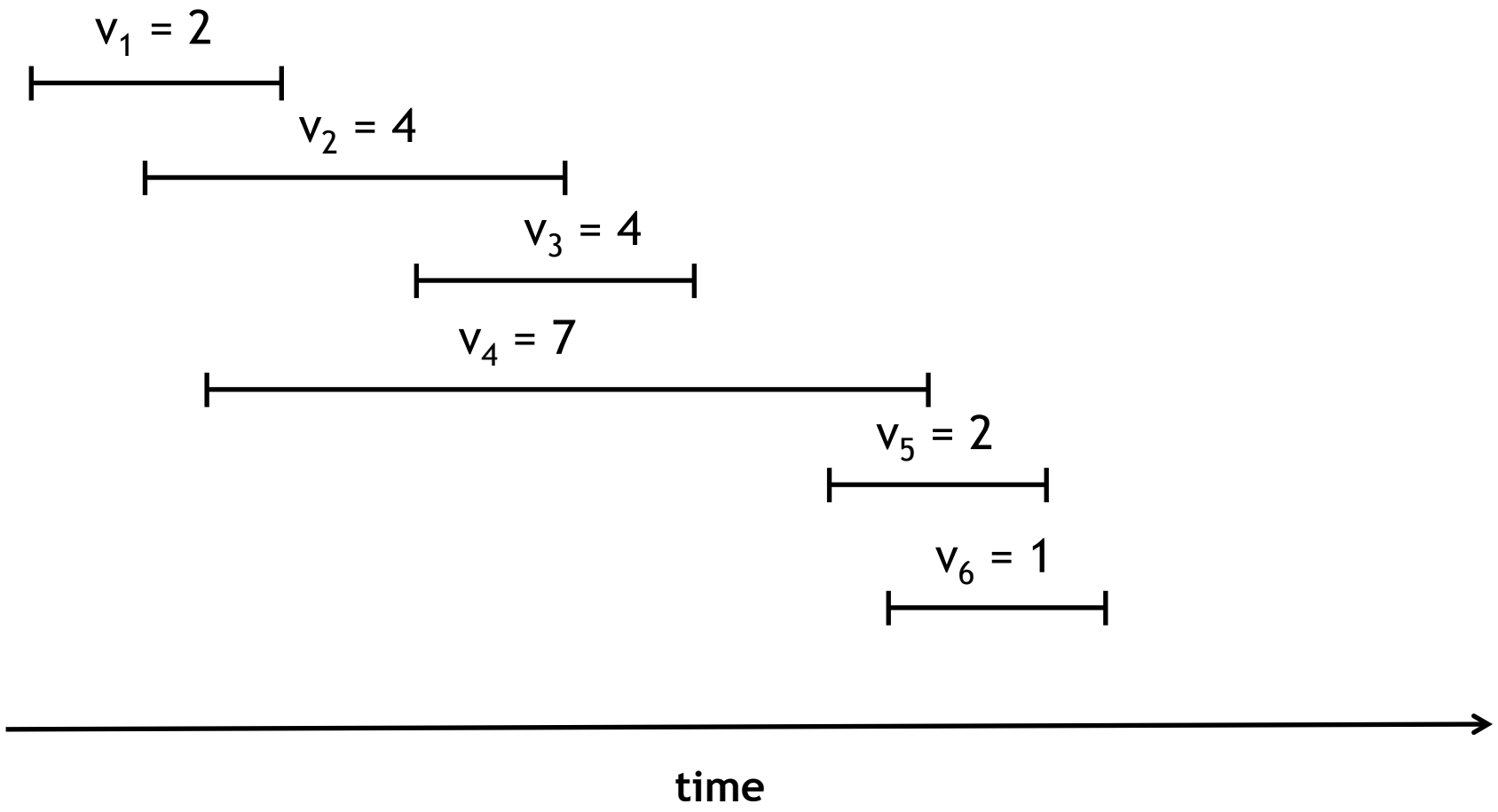


- Output: accept a set of non-overlapping requests with *max value*
- I.e: Select a set S of requests s.t.

$\forall$ (i, j) either f(i) ≤ s(j) or f(j) ≤ s(i) AND

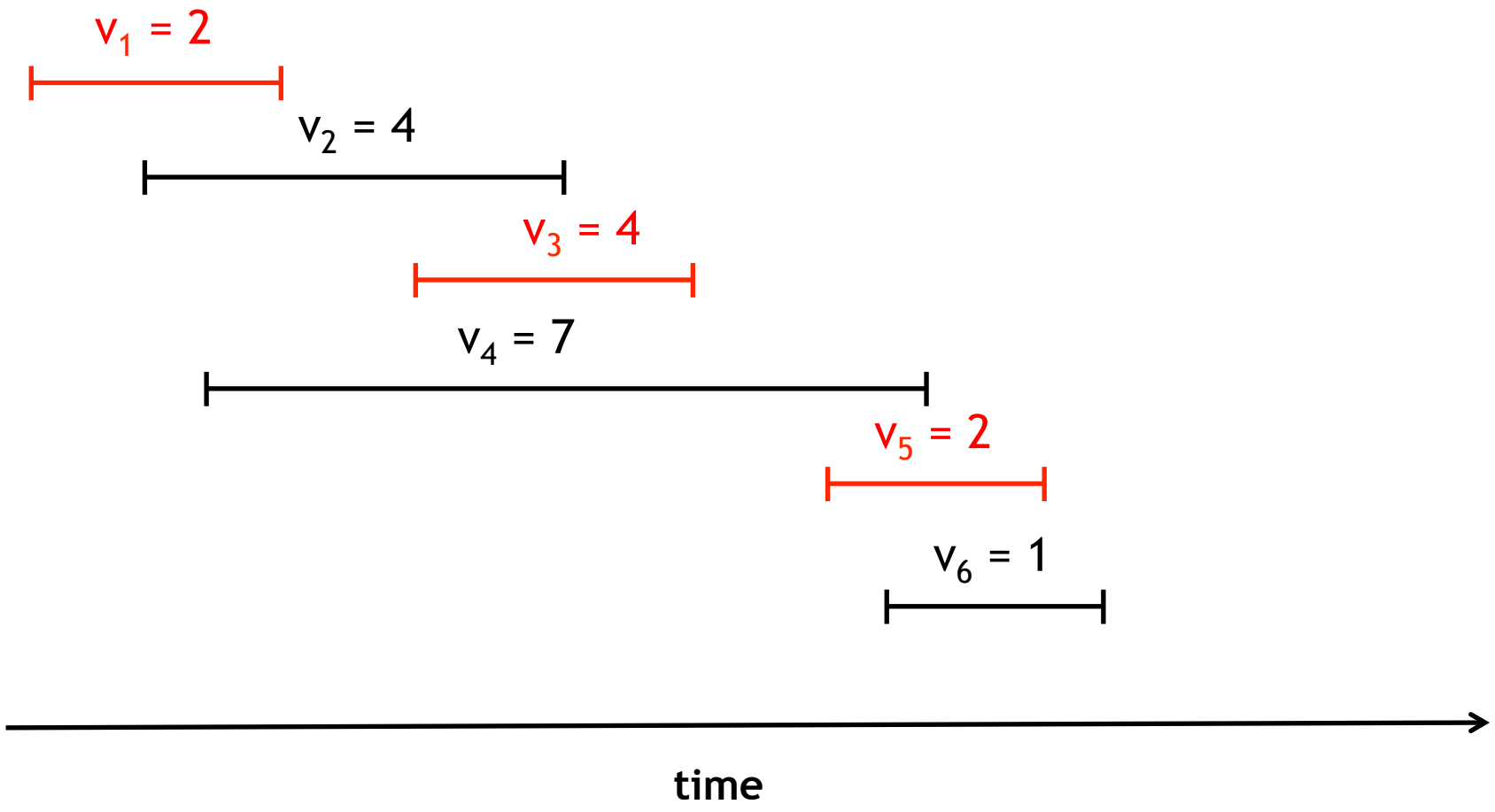
$\sum_{i \in S} v_i$ Is max over all such S

Example



OPT = 8

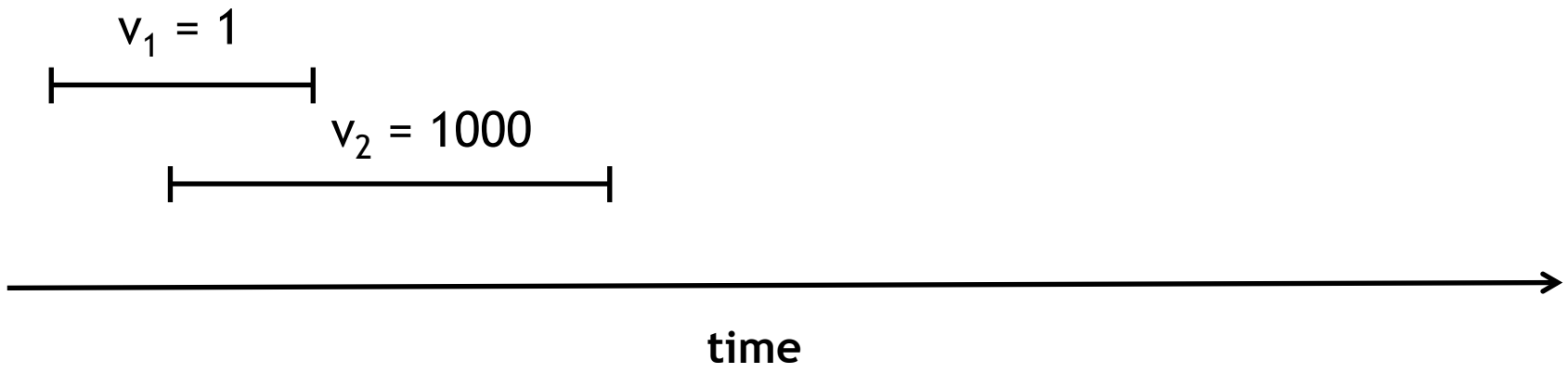
Example



OPT = 8

Greedy Earliest Finish Time Fails

- Counter example:



OPT = 1000

Greedy-Earliest-Finish-Time: 1

Recall Recipe of a DP Algorithm

1: Identify small # of subproblems.

2: quickly + correctly solve "larger" subproblems given solutions to smaller ones.

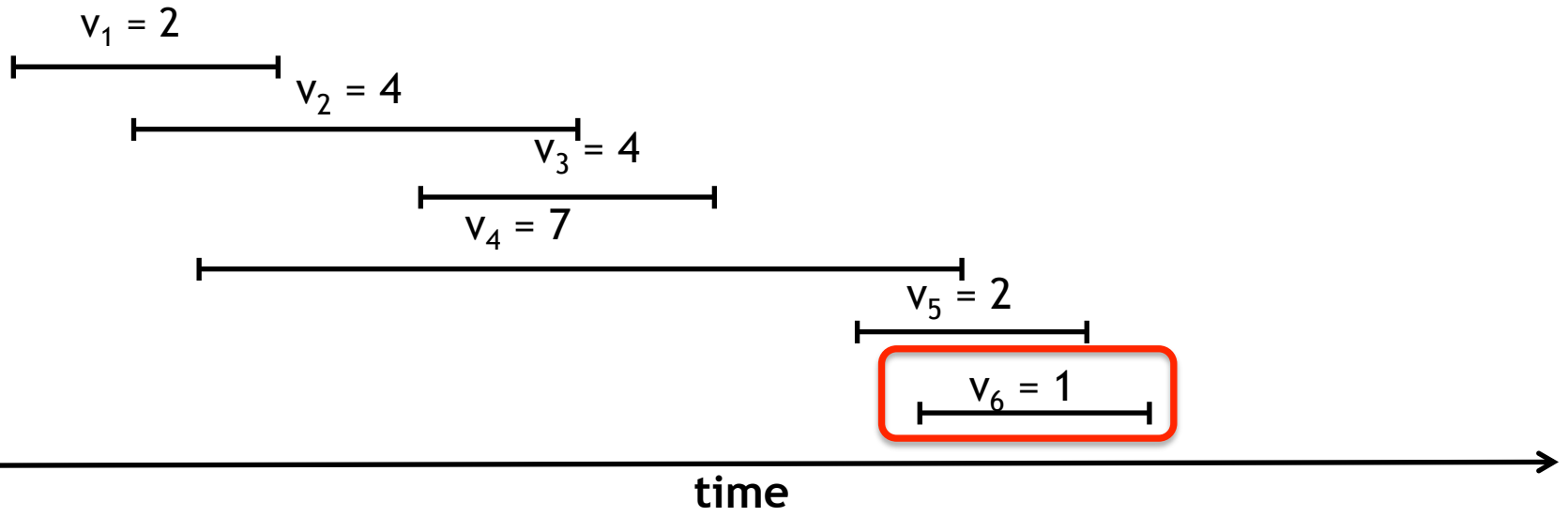
3: After solving all subproblems, quickly compute final solution

Step 1: Identifying Subproblems

- Assume w.l.o.g that r_i are ordered by non-decreasing ft.
- I.e. $f(r_1) \leq f(r_2) \leq \ldots \leq f(r_n)$
- Similar to Linear IS. Let P(k) be subproblem containing $r_1 \ldots r_k$.
- *Consider r_n. Call optimal Solution S*.*

A Claim That Doesn't Require a Proof:

There are 2 possible cases: (1) $r_n \notin S^*$ or (2): $r_n \in S^*$

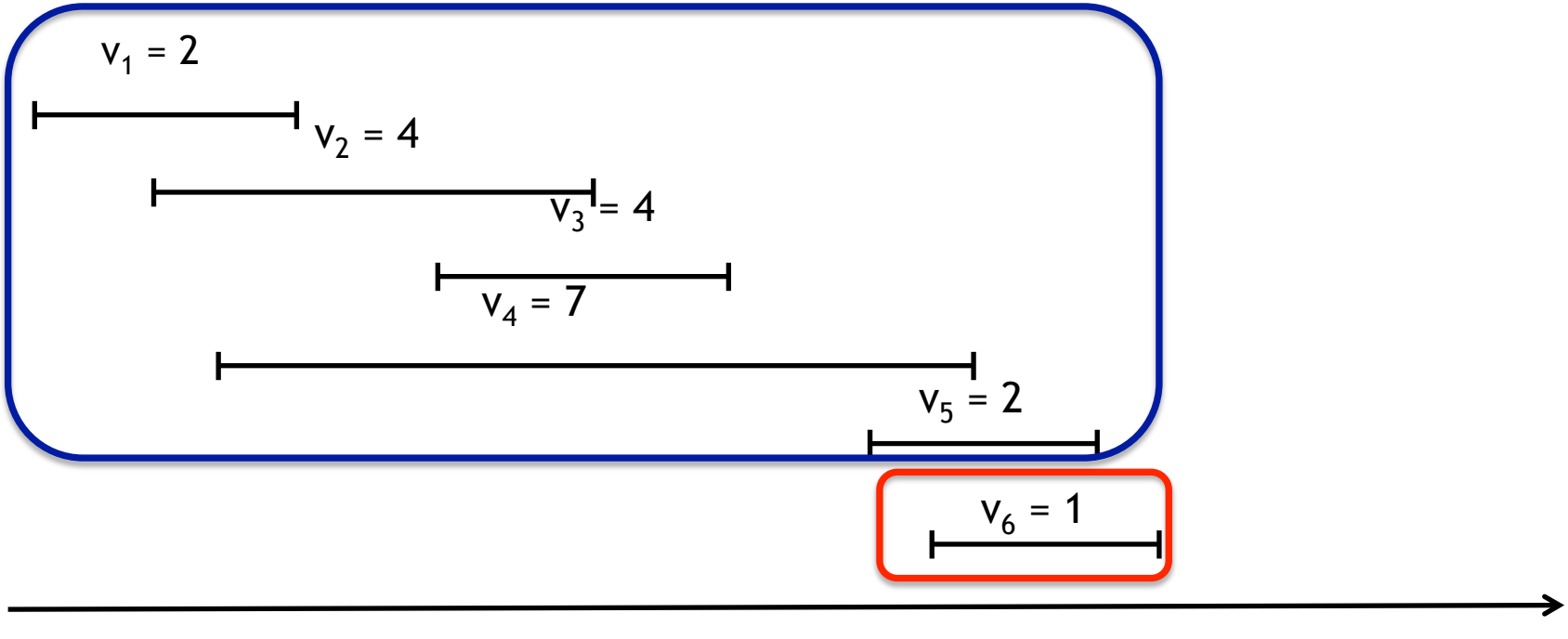


Case 1: $r_n \notin S^*$

Q: What can we assert about S* on the problem P(n-1)?

A: S* is optimum for P(n-1)

Proof: Assume $\exists$ S` in G` s.t. v(S`) > v(S*) => since S` is a valid solution in P(n), S* cannot be the max valued selection in P(n).



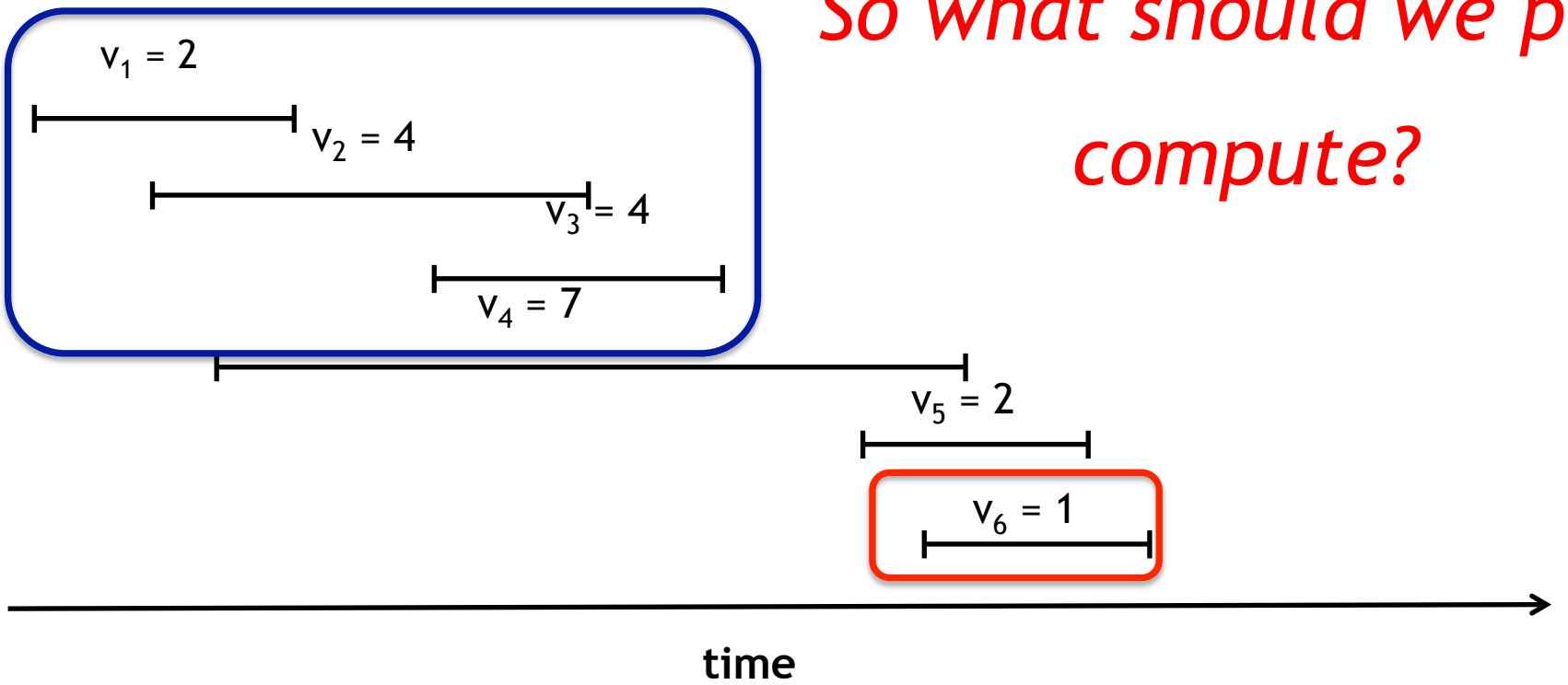
Case 2: $r_n \in S^*$

Q: What can we assert about $S^*-\{r_n\}$?

A: $S^*-\{r_n\}$ is optimum for P(i*), where i* is the largest index of a request r_{i^*} that doesn't intersect with r_n.

Why? => By contradiction (exercise)

So what should we pre-compute?



Definition z(j)

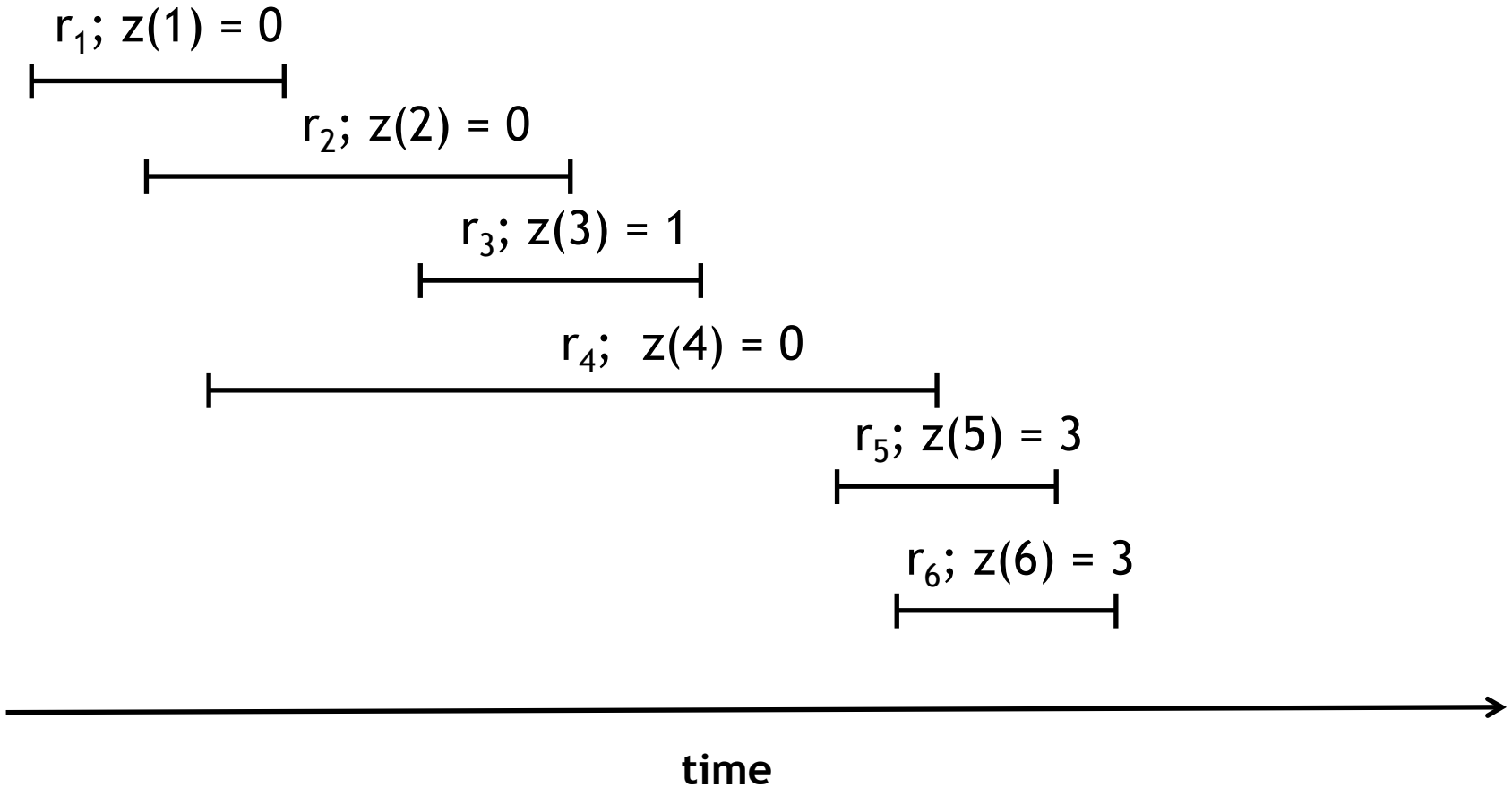
For each r_j:

Let z(j)=i be largest index of a request r_i, where i < j that doesn't intersect with r_j.

(and z(1) = 0)

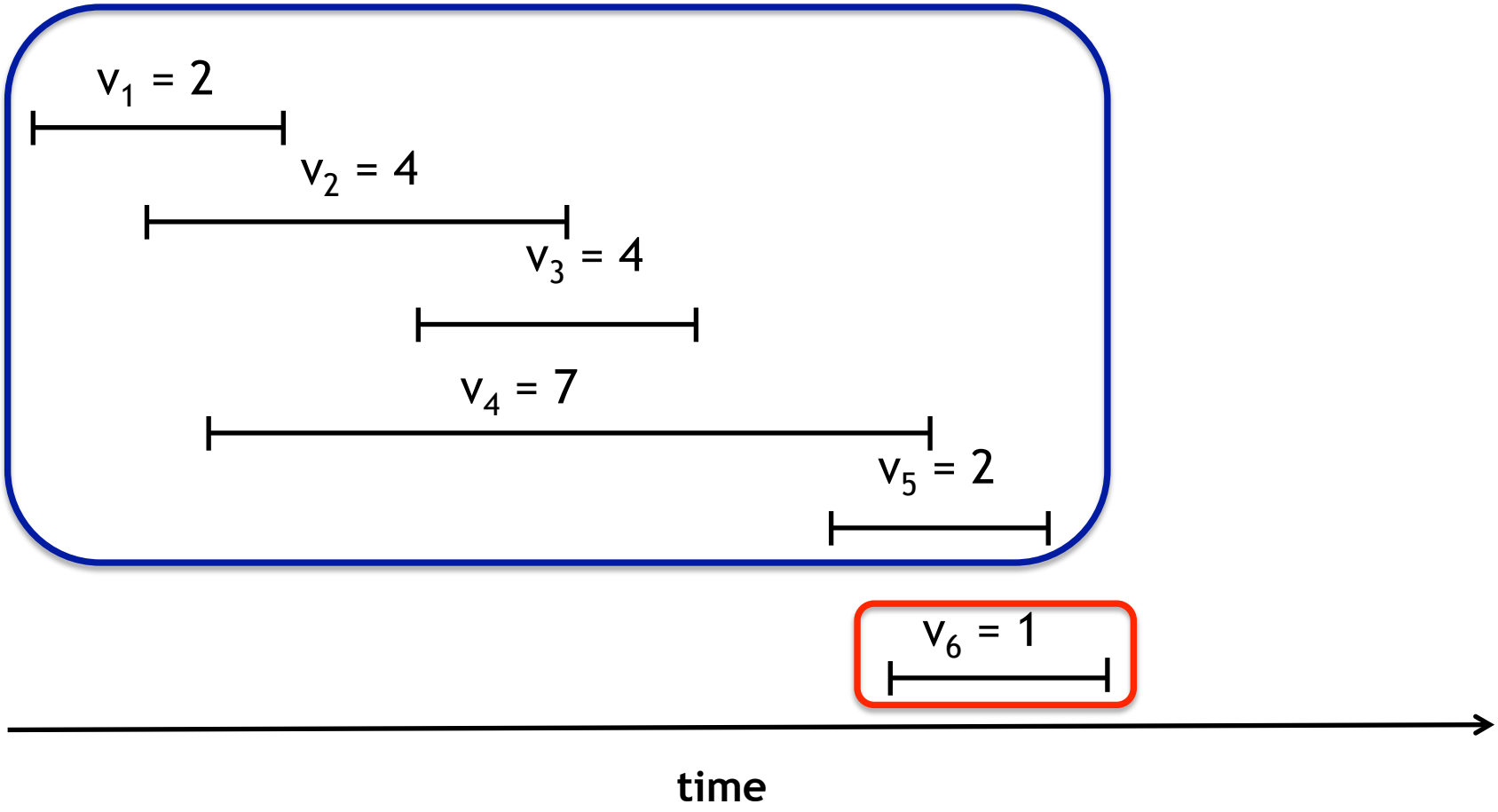
Exercise: Can compute in linear time if the requests are sorted by finish time.

Example



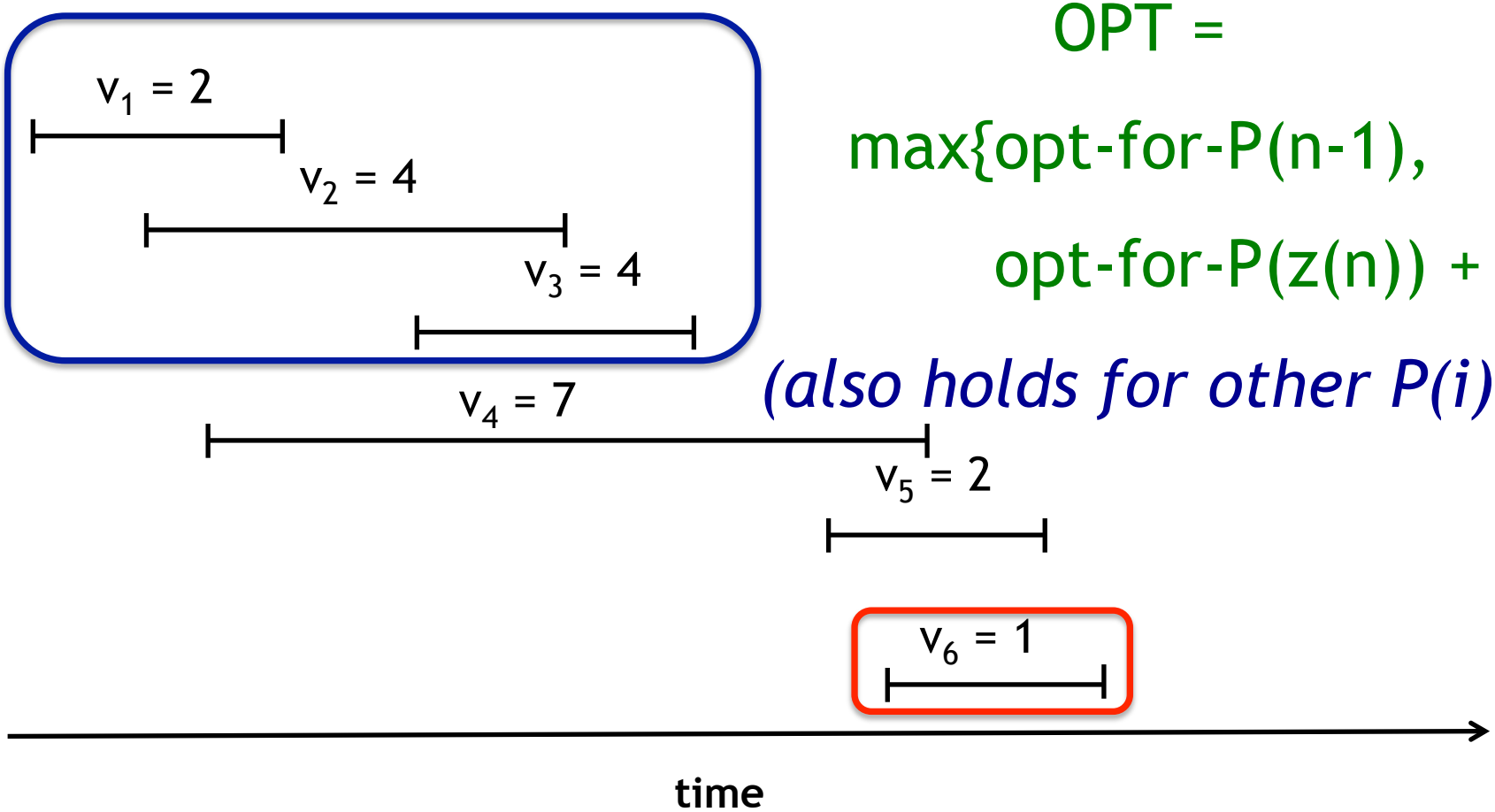
Step 2: Solve Larger Subproblems Using Solutions to Smaller Subproblems

1. $r_n \notin S^*$ => S^* is optimal for P(n-1)



Step 2: Solve Larger Subproblems Using Solutions to Smaller Subproblems

1. $r_n \notin S^* \Rightarrow S^*$ is optimal for P(n-1)
2. $r_n \in S^* \Rightarrow S^* - \{v_n\}$ is optimal for P(z(n))



OPT =

max{opt-for-P(n-1),

opt-for-P(z(n)) + v_n}

(also holds for other P(i))

DP Algorithm For Weighted Activity Selection

Let z(j) be the largest i < j, such that r_i doesn't overlap with r_j

(suppose z(j) are already computed)

Assume $f(r_1) \leq f(r_2) \leq \ldots \leq f(r_n)$ (i.e., requests are sorted)

Let A be an array of size n.

A[k] = max valued selection for requests $r_1 \ldots r_k$

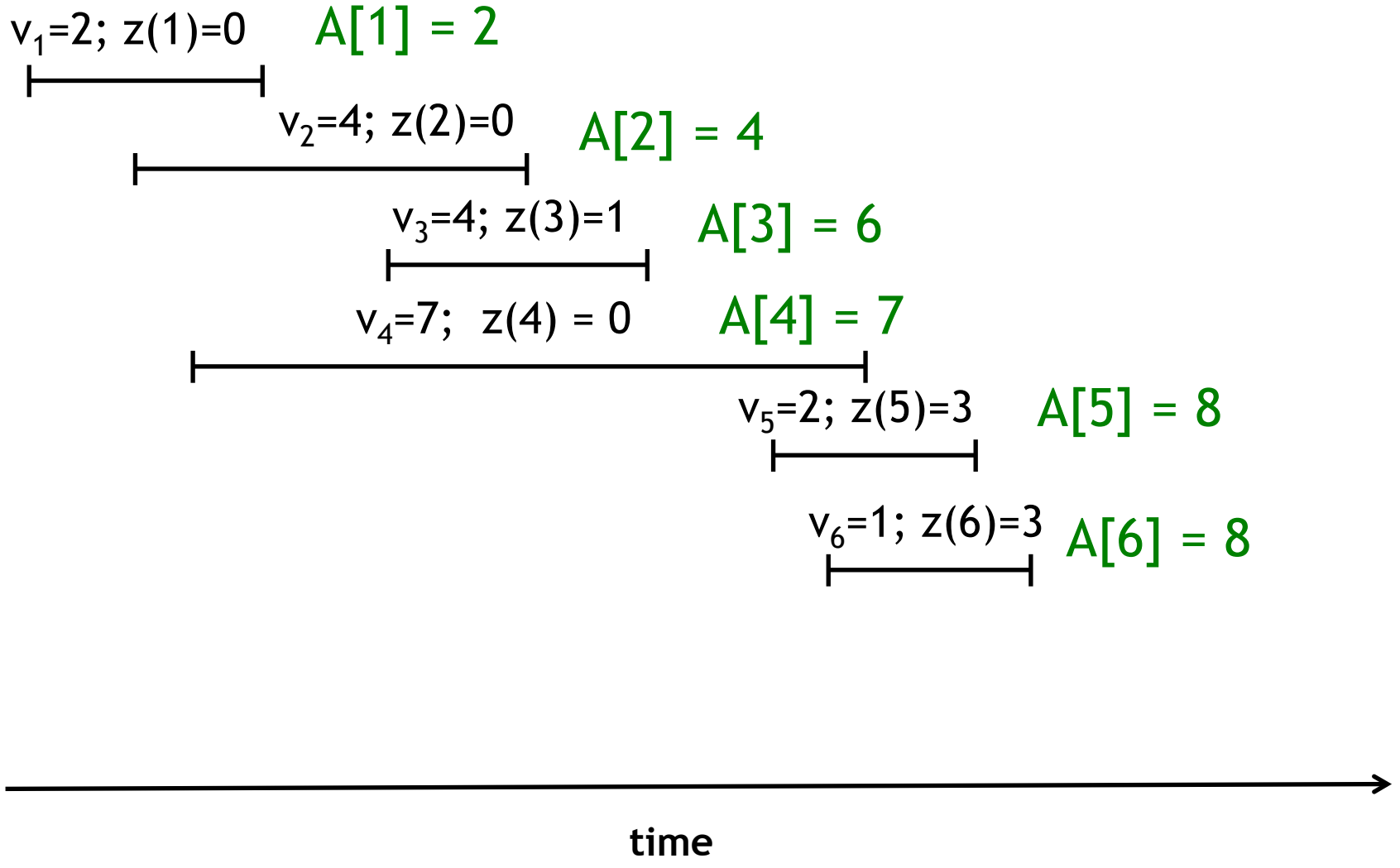
```
procedure DP-Max-Valued-Selection(r_1,…,r_n):
   Base Case: A[0] = 0
              A[1] = v_1
   for i = 2,3,…,n:
      A[i] = max{A[i-1], A[z(i)] + v_i}
   return A[n]
```

Runtime: O(n)!

(assuming r_i are sorted)

Step 3

Example


v_1=2; z(1)=0
A[1] = 2
v_2=4; z(2)=0
A[2] = 4
v_3=4; z(3)=1
A[3] = 6
v_4=7; z(4) = 0
A[4] = 7
v_5=2; z(5)=3
A[5] = 8
v_6=1; z(6)=3
A[6] = 8
time

Reconstructing the Optimal S* (1)

Let S_i be the optimal solution to P[i]. So $S^* = S_n$

Option 1: Store for each A[i] also the S_i.

=> quadratic space

Option 2: Reconstruct the solution.

Claim: $r_i \in S_i$ iff:

$$v_i + A[z(i)] \geq A[i-1]$$

Essentially iff we were in case 2 in our 2 possible worlds.

Proof: Same as before (slides 42 & 43)

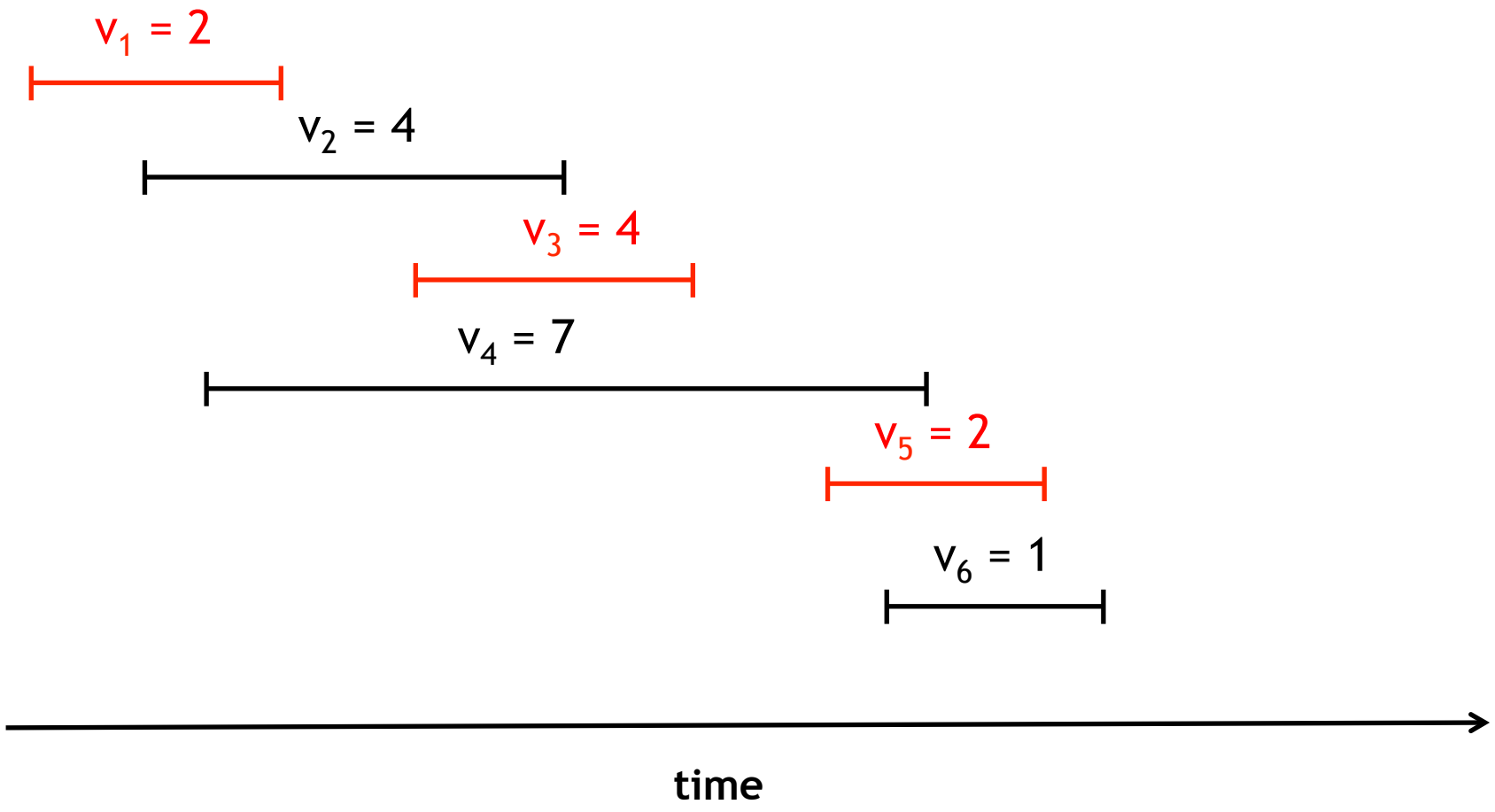
Reconstructing the Optimal IS (3)

```
procedure DP-MaxV-Sel-Reconst(r_1,…,r_n, Values, z's):
  A = DP-Max-Valued-Selection(r_1,…,r_n)
 let S = ∅
  i = n
  while i ≥ 0:
     if v_i + A[z(i)] ≥ A[i-1]:
       put r_i into S; i = z(i)
     else: i = i - 1;
  return S
```

Note: You have to start from i = n and decrease (not vice versa)

A	2	4	6	7	8	8
Values	2	4	4	7	2	1
z	0	0	1	0	3	3

Example



OPT = 8