CSE 331: Algorithms & Complexity

"Dynamic Programming"

Prof. Charlie Anne Carlson (She/Her)

Lecture 30

Wednesday Nov 12th, 2025

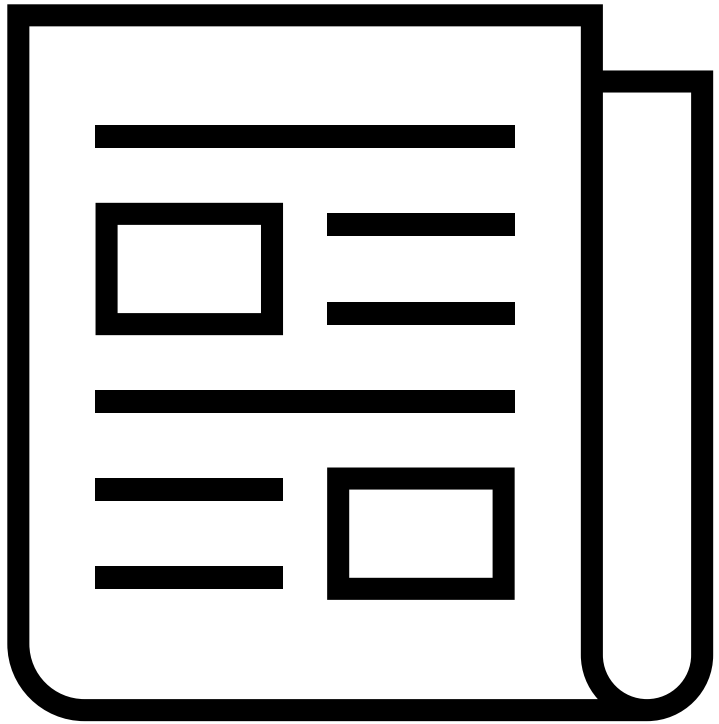
University at Buffalo®

Schedule

1. Course Updates
2. WIS
3. Dynamic Programming
    1. Optimal Substructure
    2. Overlapping Subproblems
    3. Subproblem Ordering
4. Subset Sum

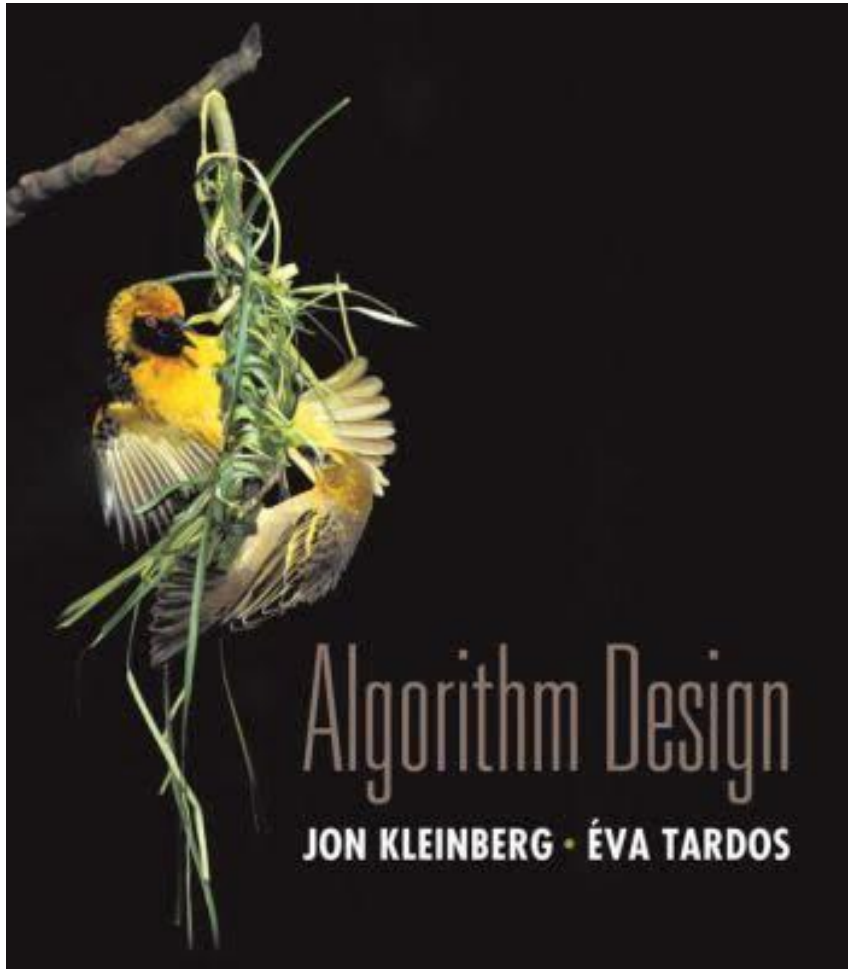
Course Updates

- HW 7 Out
  - Due November 18th
- Group Project
  - Code 3 Due November 24th
  - Reflections 3 Due December 1st
- Next Quiz is December 1st



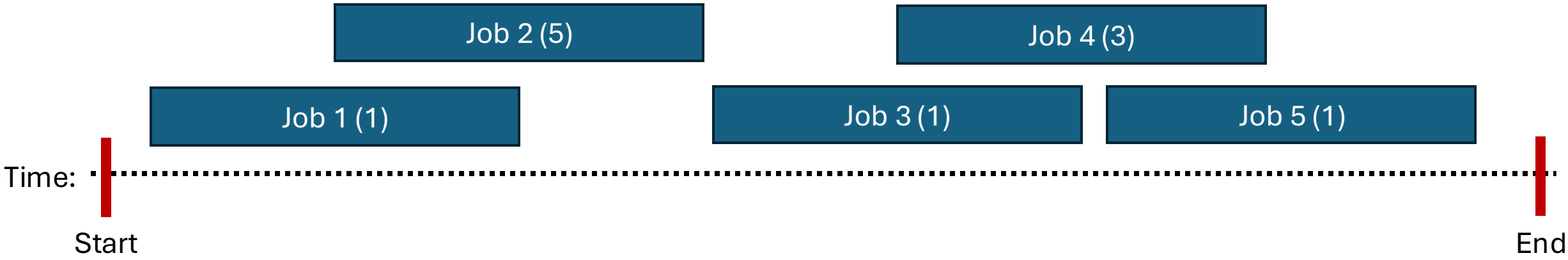
Reading

- You should have read:
  - Finished 6.1
  - Finished 6.2
- Before Next Class:
  - Finish 6.4



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.

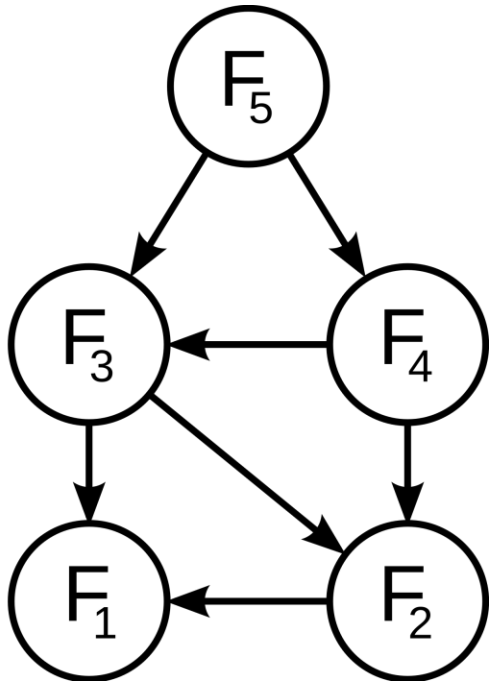


DP for WIS

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

M-Compute-Opt-Bottom-Up(j):
      For i in [j]:
            M[i] = Max(M-Compute-Opt(i-1), v[i] + M-Compute-Opt(p[i])}
      Return M[j]
```

Dynamic Programming

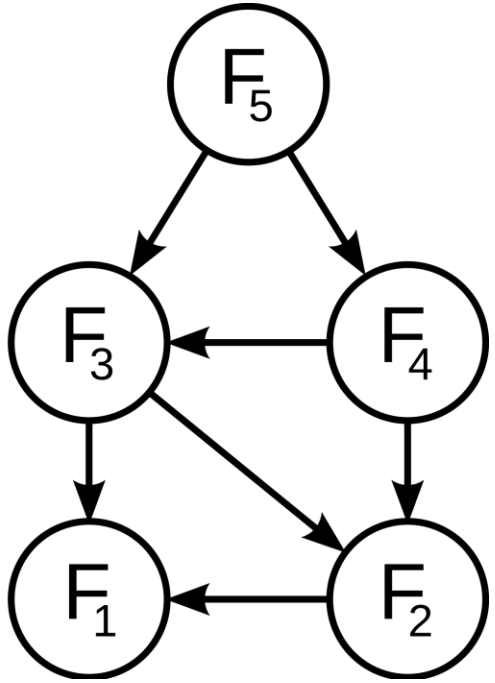
- Optimal Substructure
  - The solution to the original problem can be easily computed from the solutions to the subproblems.
- Overlapping Subproblems
  - At most polynomial subproblems to solve.
- Subproblem Ordering
  - There exists a natural order to solve subproblems without conflict.



$$F_i = \begin{cases} 0 & \text{i} = 0 \\ 1 & \text{i} = 1 \\ F_{i-1} + F_{i-2} & \text{i} \geq 2 \end{cases}$$

Dynamic Programming

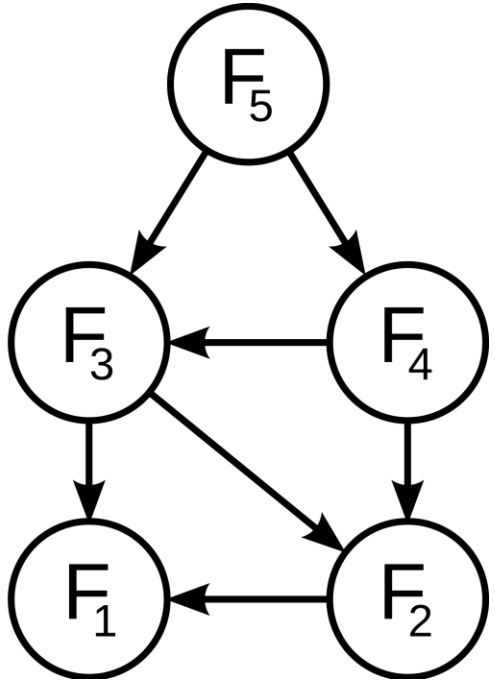
- Optimal Substructure
  - The solution to the original problem can be easily computed from the solutions to the subproblems.
- Overlapping Subproblems
  - At most polynomial subproblems to solve.
- Subproblem Ordering
  - There exists a natural order to solve subproblems without conflict.



$$F_i = \begin{cases} 0 & \text{i} = 0 \\ 1 & \text{i} = 1 \\ F_{i-1} + F_{i-2} & \text{i} \geq 2 \end{cases}$$

Dynamic Programming

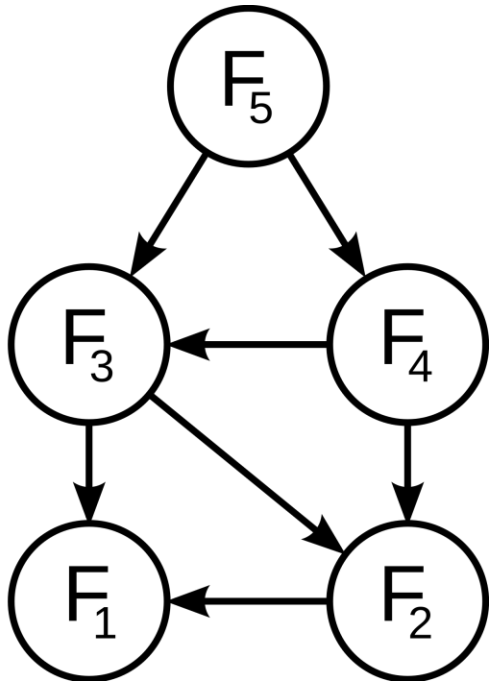
- Optimal Substructure
  - The solution to the original problem can be easily computed from the solutions to the subproblems.
- Overlapping Subproblems
  - At most polynomial subproblems to solve.
- Subproblem Ordering
  - There exists a natural order to solve subproblems without conflict.



$$F_i = \begin{cases} 0 & \text{i} = 0 \\ 1 & \text{i} = 1 \\ F_{i-1} + F_{i-2} & \text{i} \geq 2 \end{cases}$$

Dynamic Programming – Fibonacci

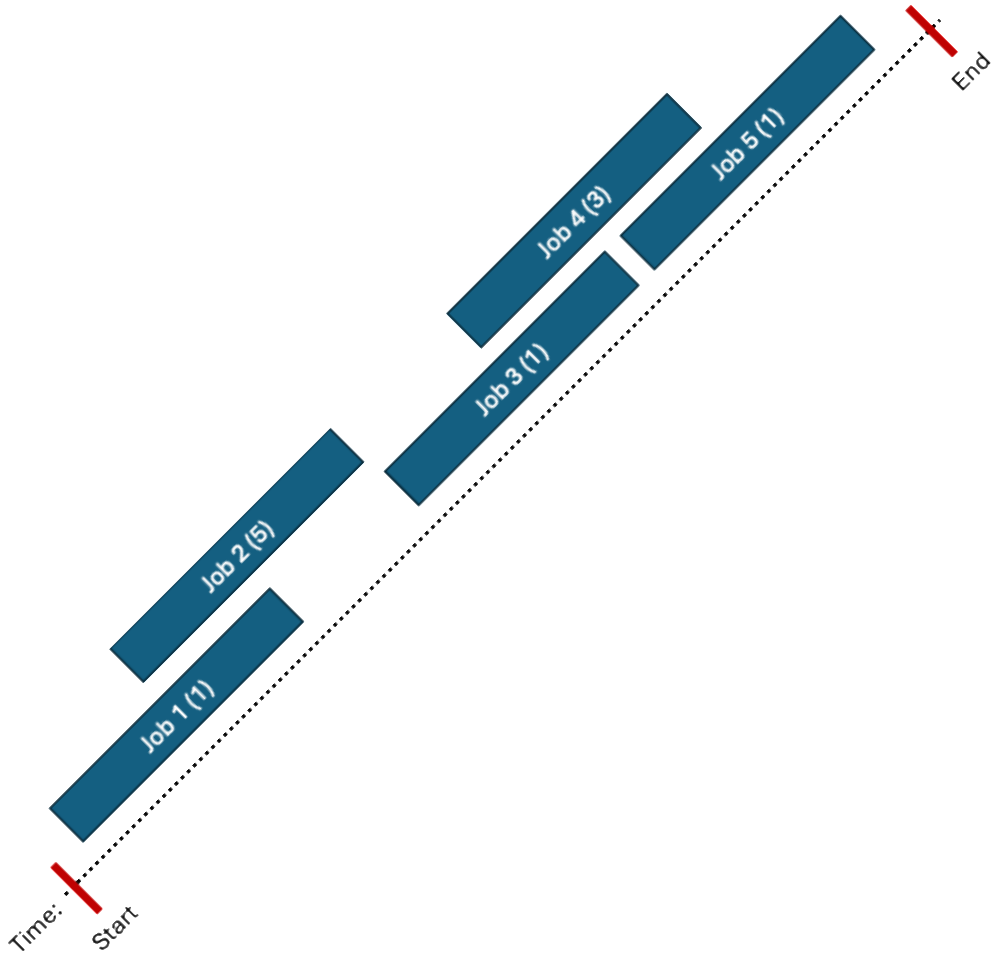
- Optimal Substructure
  - To compute F_n we can compute F_i for all $\mathrm{i} \leq \mathrm{n}$.
- Overlapping Subproblems
  - Observe that to compute F_n and F_{n-1} you need to know F_{n-2}.
  - There are at most n of them!
- Subproblem Ordering
  - Observe that you only need to know F_i for $\mathrm{i} \leq \mathrm{j}$ to compute F_j.



$$F_i = \begin{cases} 0 & \mathrm{i} = 0 \\ 1 & \mathrm{i} = 1 \\ F_{i-1} + F_{i-2} & \mathrm{i} \geq 2 \end{cases}$$

Dynamic Programming – WIS

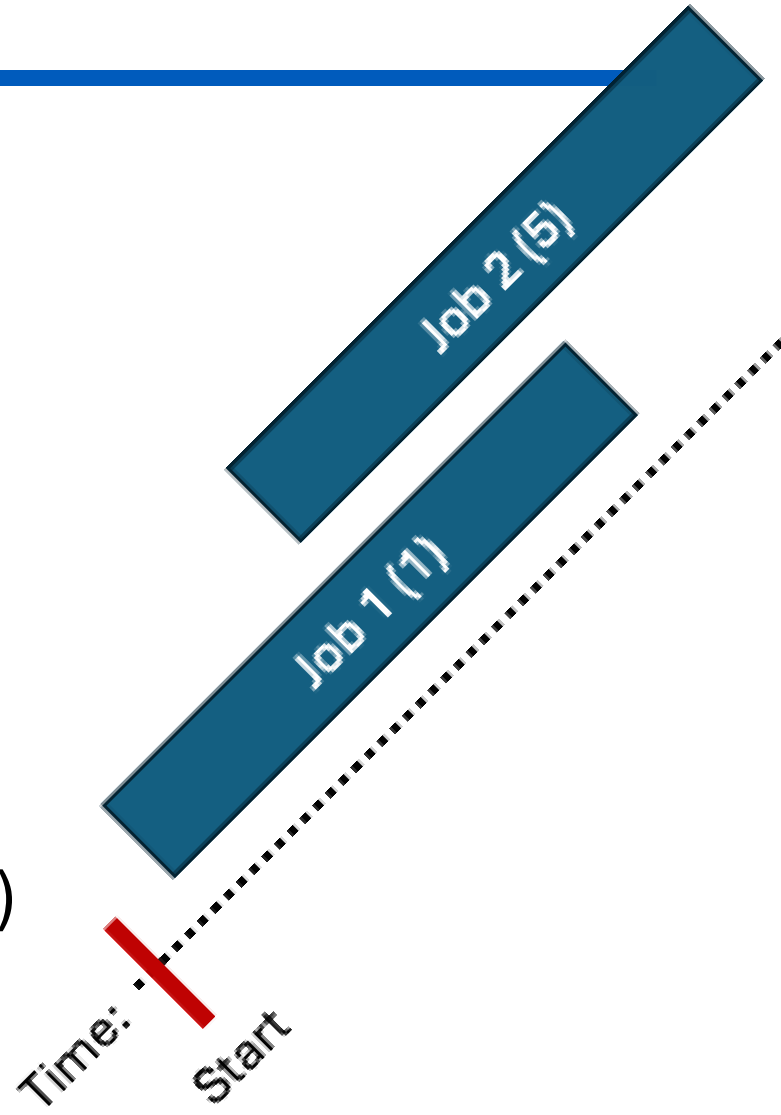
- Optimal Substructure
  - ?
- Overlapping Subproblems
  - ?
- Subproblem Ordering
  - ?



Dynamic Programming – WIS

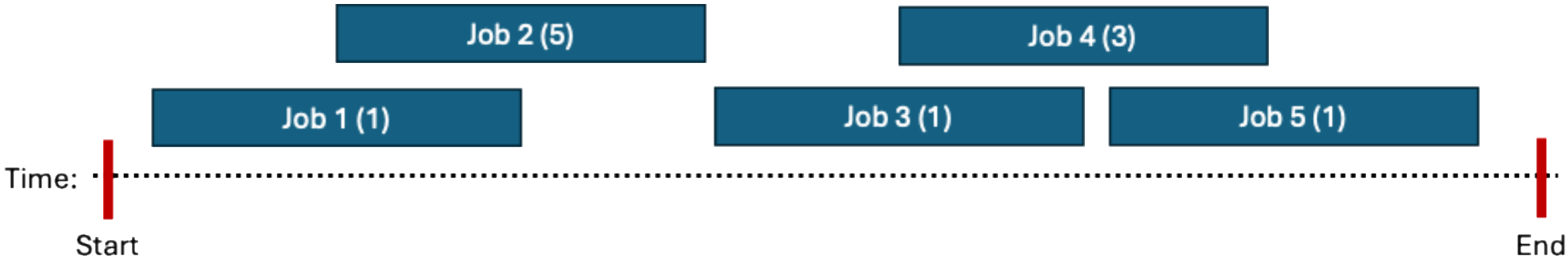
- Optimal Substructure
  - Define Subproblems:
    - Let OPT(i) be the best job schedule using only first i jobs.
  - Describe Using Subproblems:
    - Given OPT(i) for all i < n, we can compute:

$$OPT(n) = MAX(OPT(n-1), v[n] + OPT(P[n]))$$



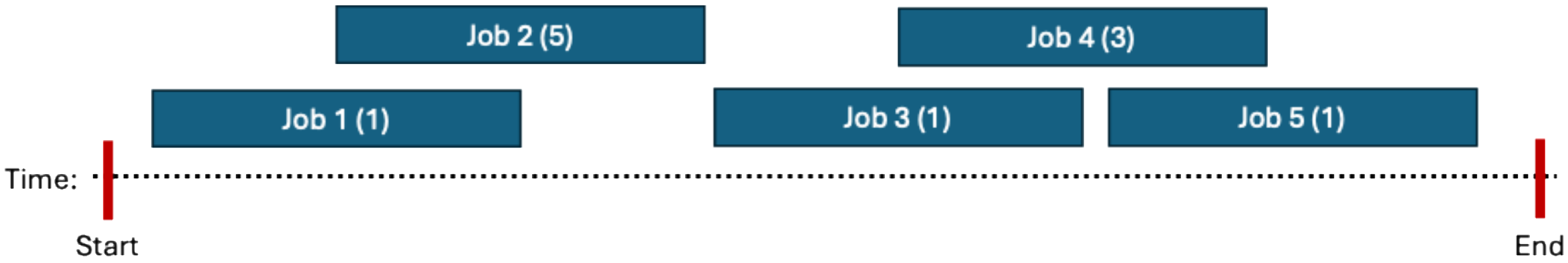
Dynamic Programming – WIS

- Overlapping Subproblems
  - In many instances, it is easy to see that you will need OPT(i) multiple times.
  - You only ever need to consider i from 1 to n.
  - E.g.: You need OPT(2) to compute OPT(3) and OPT(4) in the following example:



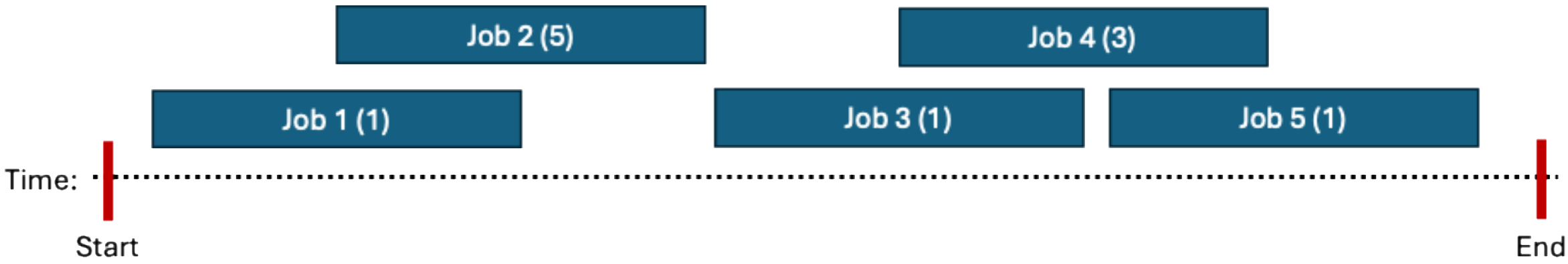
Dynamic Programming – WIS

- Subproblem Ordering
  - If you know OPT(i) for all $i < j$ then you can compute OPT(j) using substructure from before.



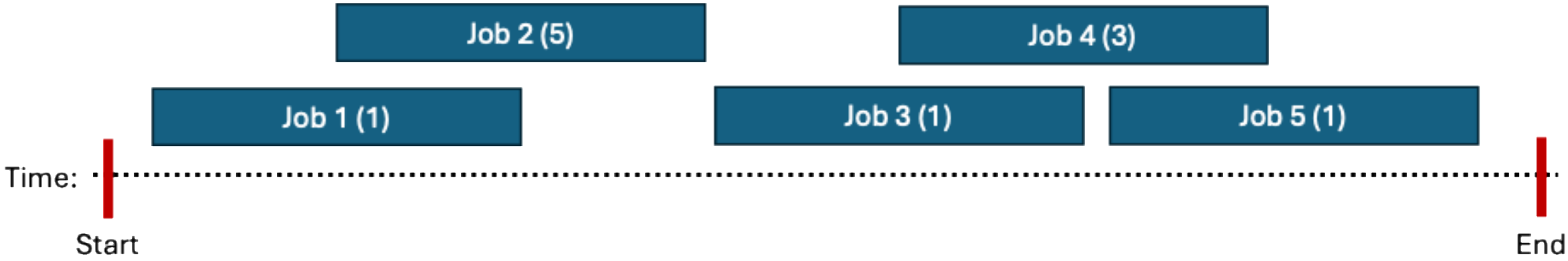
Dynamic Programming – Tips

- Describe subproblems in English:
  - “the optimal solution considering only the first i jobs.”
- Make note of the format of subproblems:
  - Each subproblem was of the same type, WIS.



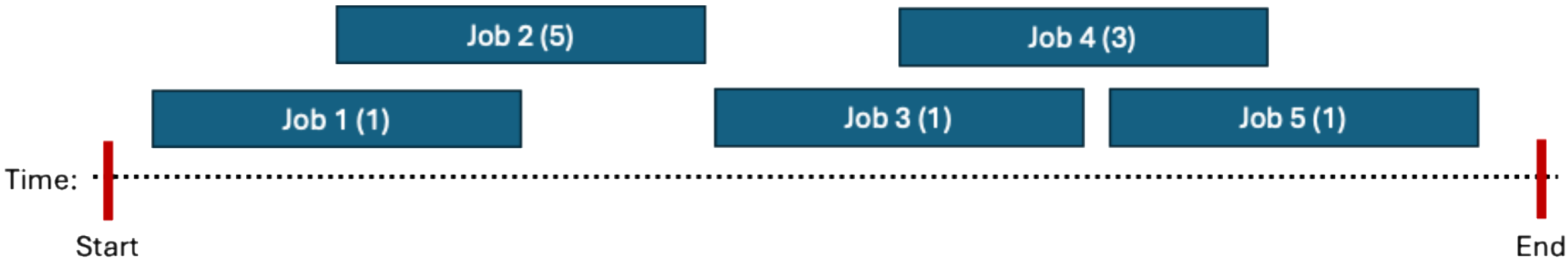
Dynamic Programming – Tips

- Describe how solving all the subproblems gets you the solution:
  - “Once we have computed OPT(i) for all i, the answer to the problem is OPT(i).”
- It is okay to do some preprocessing before you solve the problem.



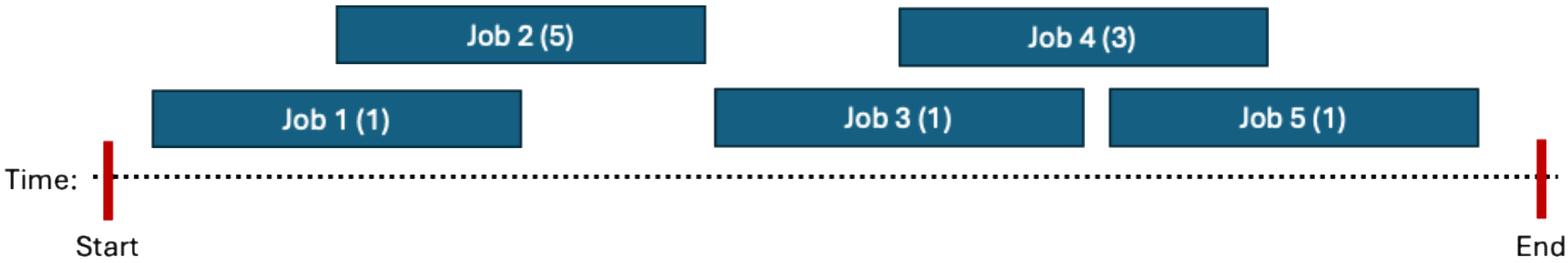
Dynamic Programming – Tips

- Make sure you consider the order of solving subproblems carefully.
  - We will see in our next problem a case where you have more than one variable.



Dynamic Programming – Tips

- Don’t forget the base case!
  - Often you need to initialize the array/matrix/tree/graph with some values or you won't be able to fill in the next case!



Dynamic Programming vs Divide & Conquer

- **Question**: What happens if you have optimal substructure, but you don’t have a lot of overlapping subproblems?

Dynamic Programming vs Divide & Conquer

- **Question**: What happens if you have optimal substructure, but you don't have a lot of overlapping subproblems?
- **Answer**:
  - If you have few subproblems, you might get a good algorithm that is divide & conquer, decrease & conquer, or even brute force.
    - E.g. Mergesort
  - If you have many subproblems than you may not get an efficient algorithm.

Dynamic Programming Bottom-Up/Iterative

- Describe and justify the subproblems.
  - Give a recurrence for the subproblems and specify where to find the final solution.
  - Determine the natural order to solve the subproblems.
- Decide on a data structure to hold the solutions
  - E.g. We used an array for WIS
- Write the code to fill in the data structure based on the natural order you found.

DP for WIS

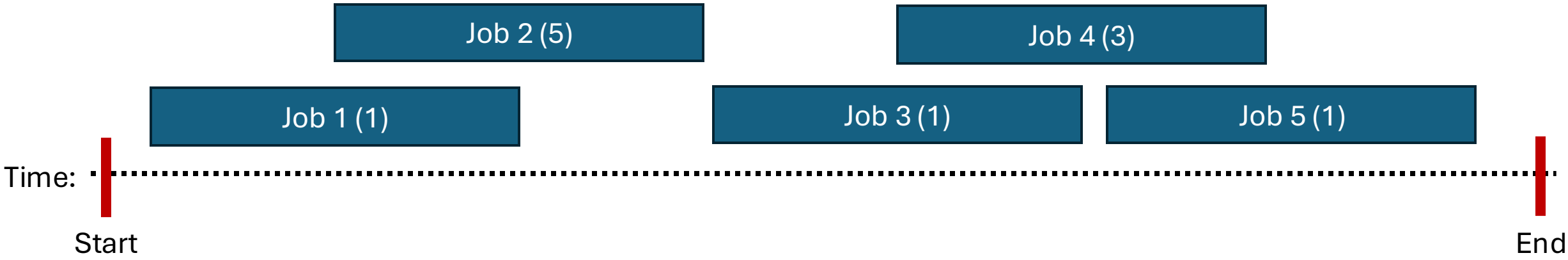
```
Brute-Force(L):
      Sort L by job finish times.
      Compute p[i] for each i using binary search.
      Global M = [], M[0] = 0
      Return Compute-Opt-?(n)

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

M-Compute-Opt-Bottom-Up(j):
      For i in [j]:
            M[i] = Max(M-Compute-Opt(i-1), v[i] + M-Compute-Opt(p[i])}
      Return M[j]
```

Dynamic Programming Choices

- Often the key to writing a dynamic programming algorithm is identifying a recurrence.
  - To find a recurrence, you often want to find a choice.
    - E.g. “You either take the job i or you don’t.”
  - Once you’ve found your choice, you can then describe a subproblem based on the outcome of all choices.
    - E.g. “If you take it, you need to consider removing all conflicting jobs. If you don’t, you don’t need to consider it anymore.”
- Sometimes the subproblem is natural and finding the recurrence is the hard part.

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.



Subset Sum

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a weight w_i
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

Subset Sum

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

- **Input**: A list of n tasks {1,…,n} you want to schedule on a server.
  - You are allowed to use at most W cycles.
  - Each item i has a weight w_i that represents needed cycles.
- **Goal**: Find the subset S of tasks such that you don't use more than your maximum number of cycles, but you get as much work done as possible, i.e., $\sum_{i \in S} w_i \leq W$.

Subset Sum Notation & Observations

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

- For any set $S \subseteq [n]$, let $w(S) = \sum_{i \in S} w_i$.
  - Note that $w(\emptyset) = 0$.
  - Also note if $w([n]) \leq W$, then the answer is [n].
  - Finally note if $w_i > W$ for all i, then answer is $\emptyset$.

Subset Sum Subproblems

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

Idea: Let OPT(i) be the optimal solution only considering the first i items.

Subset Sum Subproblems

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

Idea: Let OPT(i) be the optimal solution only considering the first i items.

Question: How can you solve the original problem if you find OPT(i) for i < n?

Subset Sum Example

Idea: Let OPT(i) be the optimal solution only considering the first i items.

Example:

- Item values = [1,4,7,2,5,12,14]
- Bound = 10

Observations:

- You can't decide if you want to take item with value 5 based on the optimal solution to [1,4,7,2] with bound 10.
  - The optimal there is 9 but it can be 5.
- If you don't keep track of what you've used, you may overuse.

Subset Sum Subproblems

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

Idea: Let OPT(i) be the optimal solution only considering the first i items.

Answer: It doesn't seem possible as you might use too much weight.

Subset Sum Subproblems

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

Idea: Let OPT(W') be the optimal with bound W'.

Question: How can you solve the original problem if you find OPT(W') for W' < W?

Subset Sum Example

Idea: Let OPT(W') be the optimal with bound W'.

Example:

- Item values = [4,7,2,5,12,14]
- Bound = 10

Observations:

- You can't decide if you want to take item with value 5 based on the optimal solution to [4,7,2,5,12,14] with bound 5.
  - The optimal there is 5 but it shouldn't be combined with 5 again.
  - That is, if you aren't keep track of the

Subset Sum Subproblems

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

Idea: Let OPT(W') be the optimal with bound W'.

Answer: It doesn't seem possible as you might use the same item multiple times.

Subset Sum Subproblems

- **Input**: A list of n items {1,…,n} and a bound W.
  - Each item i has a non-negative weight w_i.
- **Goal**: Find the max-weight subset S such that $\sum_{i \in S} w_i \leq W$.

Idea: Let OPT(i, W') be the optimal with only first i items and bound of W'.

Now we can describe the "choice" of using an item or not using it.

Subset Sum Optimal Substructure

Idea: Let OPT(i, W') be the optimal with only first i items and bound of W'.

If i = 0 then

OPT(i,W') = 0

If $W' > w_i$ then

OPT(i,W') = max{w_i + OPT(i-1,W'- w_i}, OPT(i-1,W')}

Otherwise

OPT(i,W') = OPT(i-1,W')

Subset Sum Problem Overlap

It is believable that there will be instances where we have a lot of problem overlap.

Example:

- Item values = [4,7,2,5,2,12,14]
- Bound = 20

There are two different ways to eventually ask about OPT(4,6): Picking 14 or pick 12 and 2.

Subset Sum Problem Count

There will be n choices for i and W choices for W’ (provided everything is an integer).

Example:

- Item values = [4,7,2,5,2,12,14]
- Bound = 20

You only need to consider (i,20), (i,19), …, (i,0) for each i.

Subset Sum Ordering Subproblems

Idea: Let OPT(i, W') be the optimal with only first i items and bound of W'.

If i = 0 then

OPT(i,W') = 0

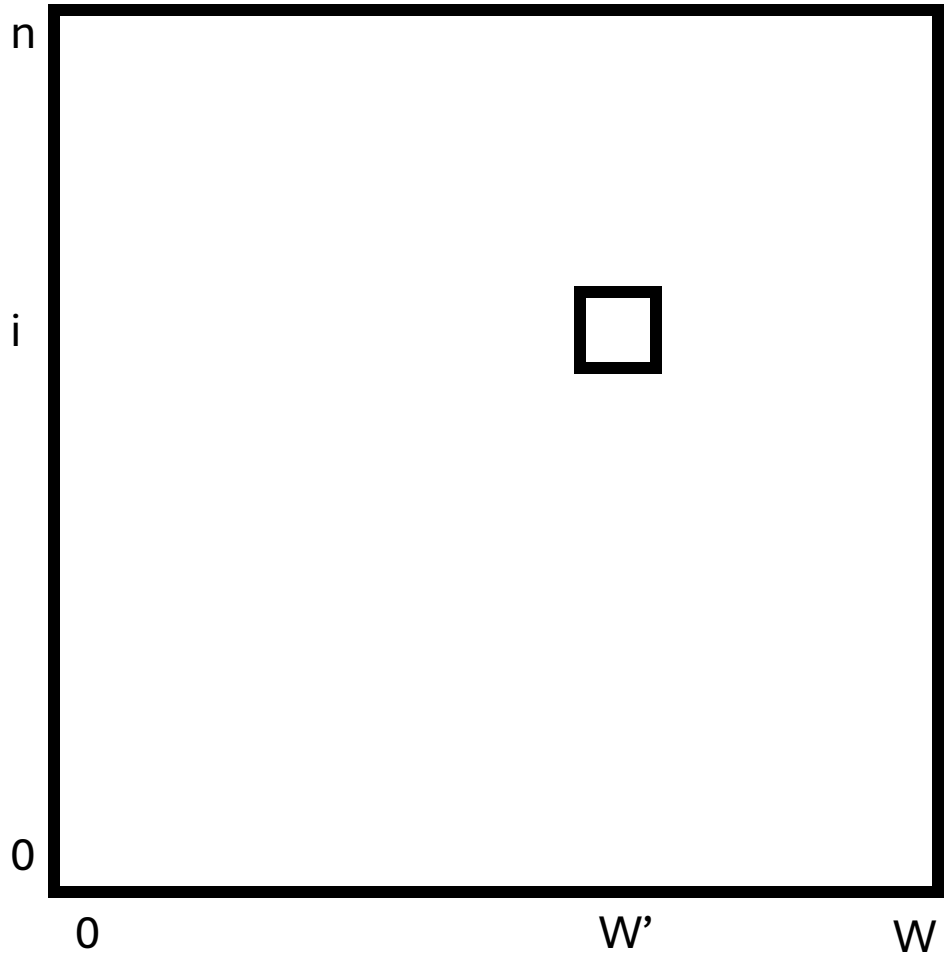
If $W' > w_i$ then

OPT(i,W') = max{w_i + OPT(i-1,W'- w_i}, OPT(i-1,W')}

Otherwise

OPT(i,W') = OPT(i-1,W')

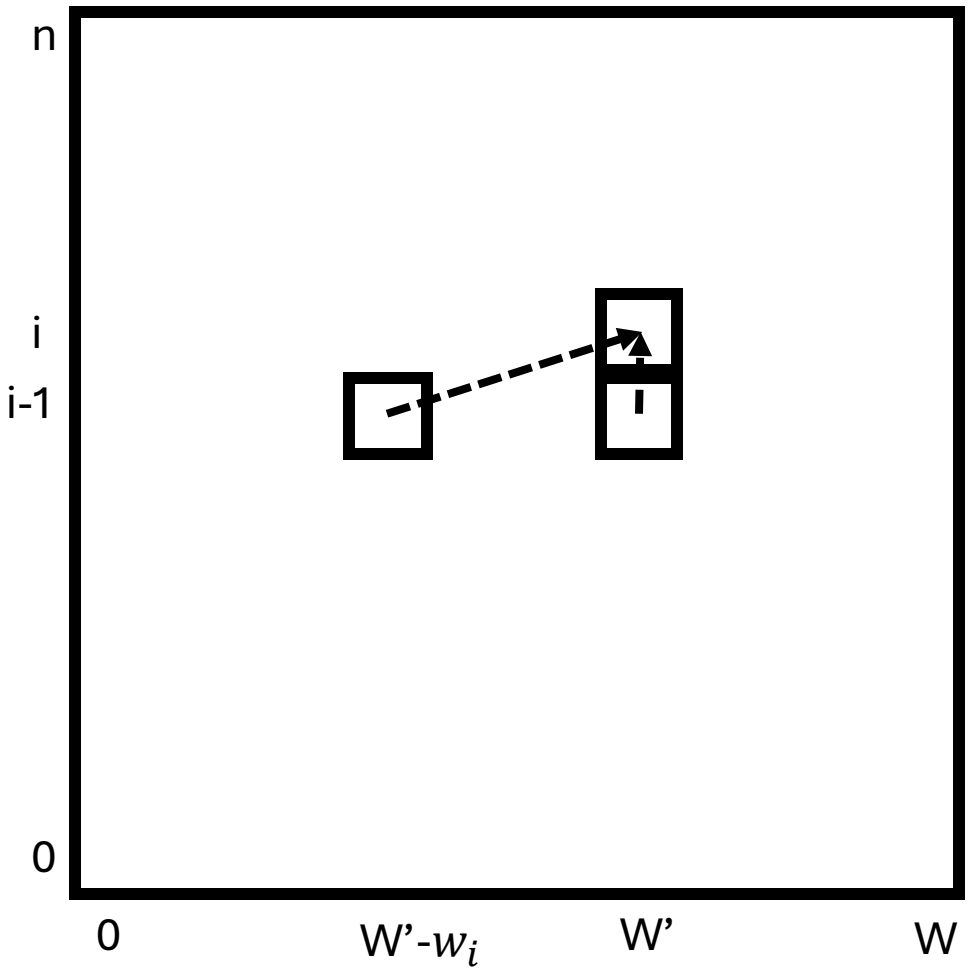
Subset Sum Ordering Subproblems



OPT(i,W') = max{w_i + OPT(i-1,W'- w_i}, OPT(i-1,W')}

Subset Sum Ordering Subproblems

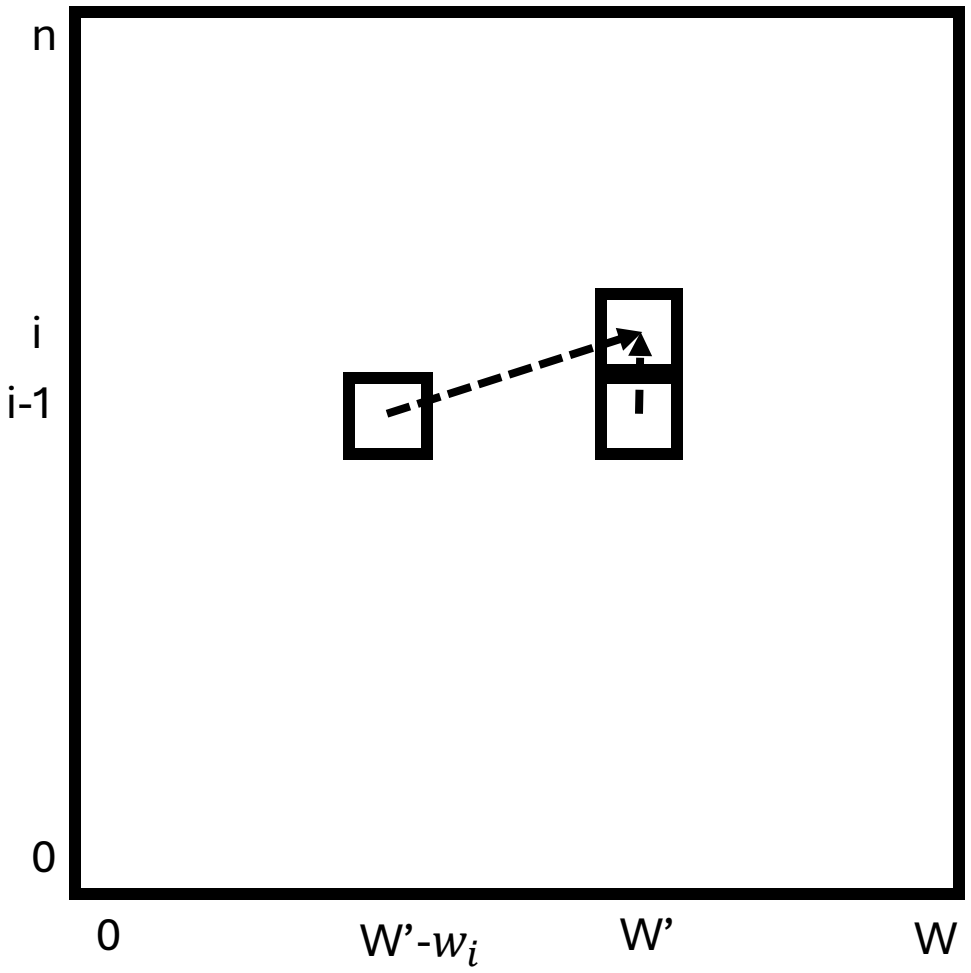
We need to know for smaller W’ and smaller i.



OPT(i,W’) = max{w_i + OPT(i-1,W’- w_i}, OPT(i-1,W’)}

Subset Sum Ordering Subproblems

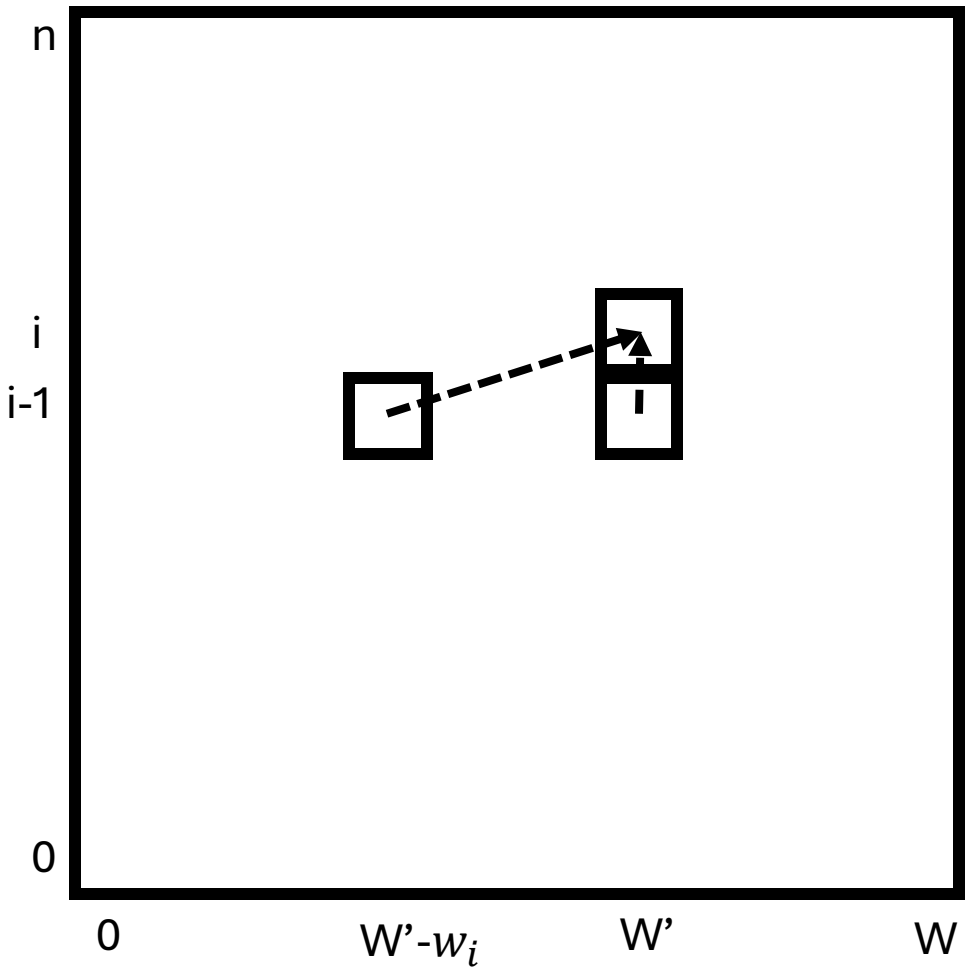
Question:
How do we fill?



OPT(i,W') = max{w_i + OPT(i-1,W'- w_i}, OPT(i-1,W')}

Subset Sum Ordering Subproblems

Answer:
Out for loop will be for i and inner for loop will be for W'.



Fill first layer with 0.

OPT(i,W') = max{w_i + OPT(i-1,W'- w_i}, OPT(i-1,W')}