CSE 331: Algorithms & Complexity

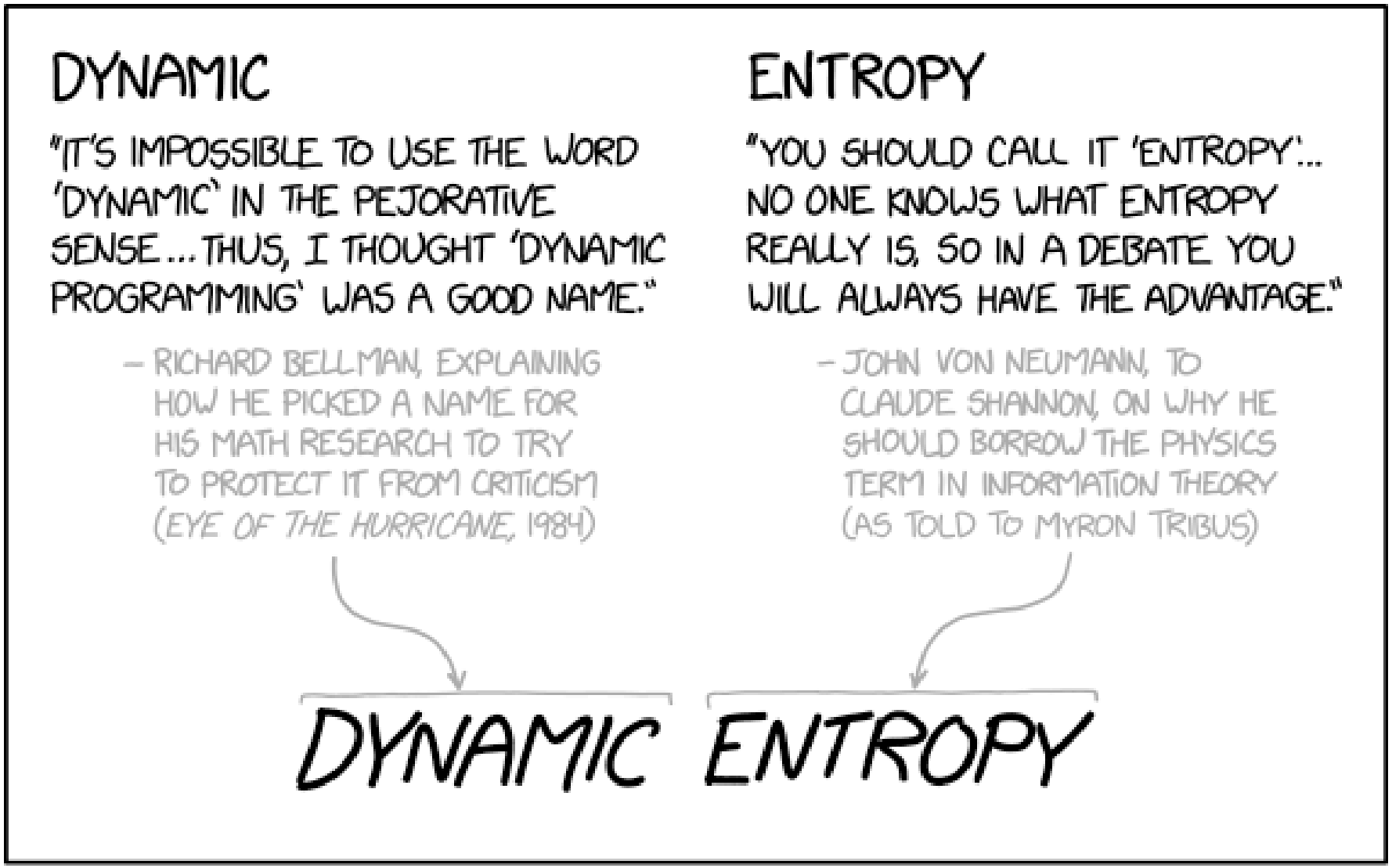
"Top-Down DP"

Prof. Charlie Anne Carlson (She/Her)

Lecture 29

Monday Nov 10th, 2025

University at Buffalo®



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

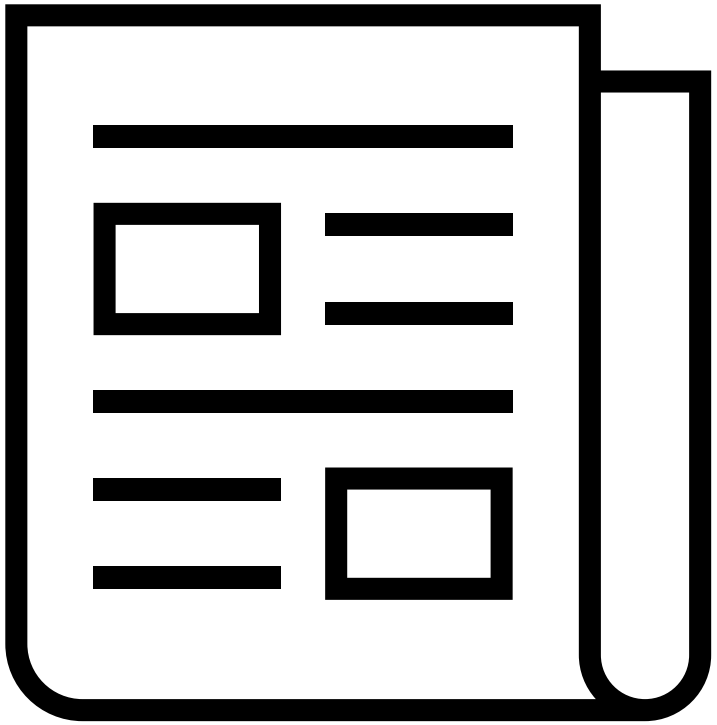
https://xkcd.com/2318/

Schedule

1. Course Updates
2. Weighted Interval Scheduling
3. Top-Down
4. Recovering Solution
5. Bottom-Up

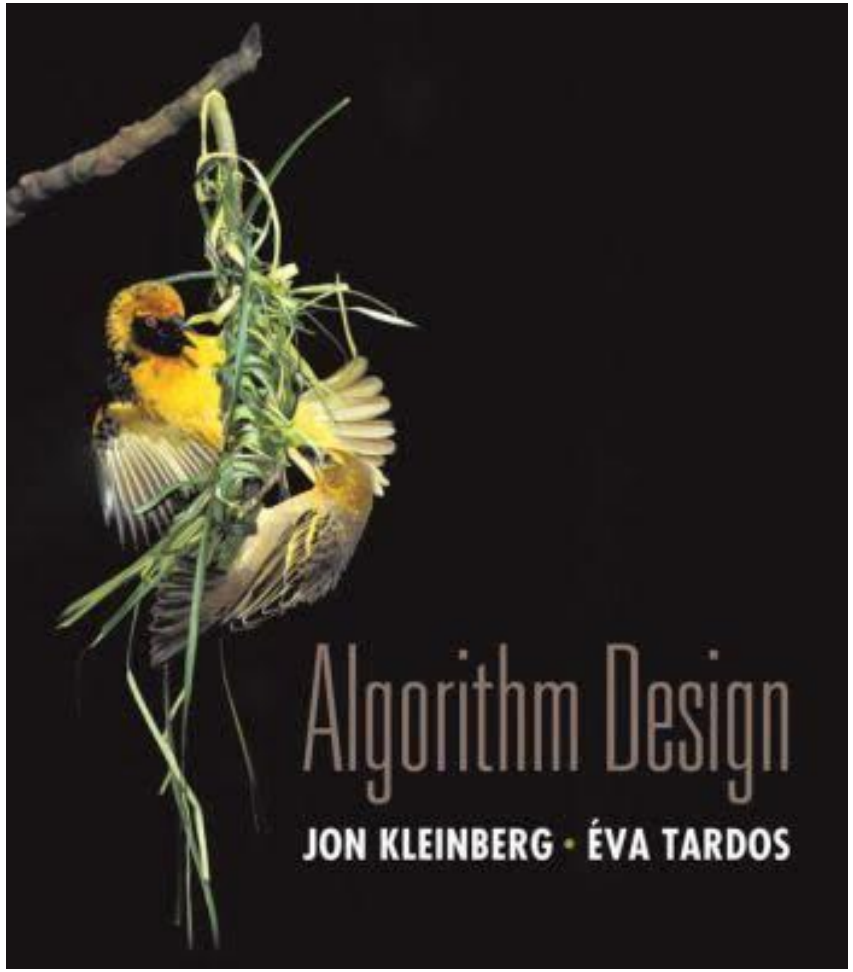
Course Updates

- HW 6 Due Tomorrow
- HW 7 Out Tomorrow
  - Due November 18th
- Group Project
  - Code 3 Due November 24th
  - Reflections 3 Due December 1st
- Next Quiz is December 1st



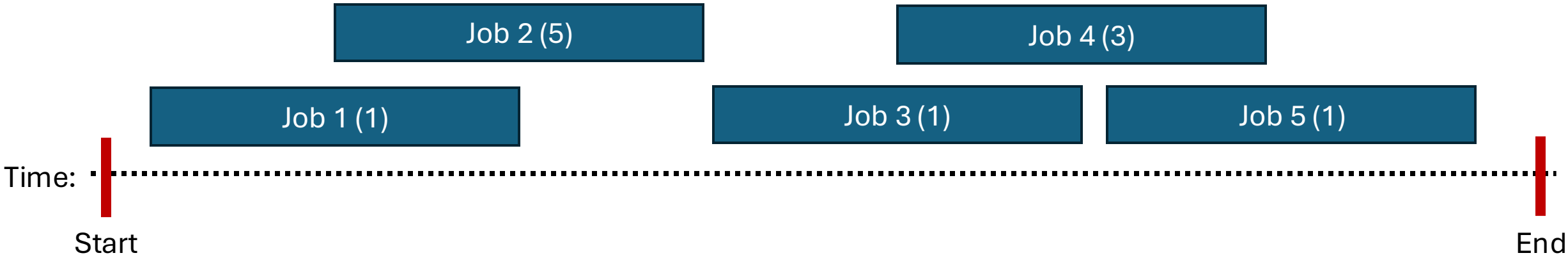
Reading

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



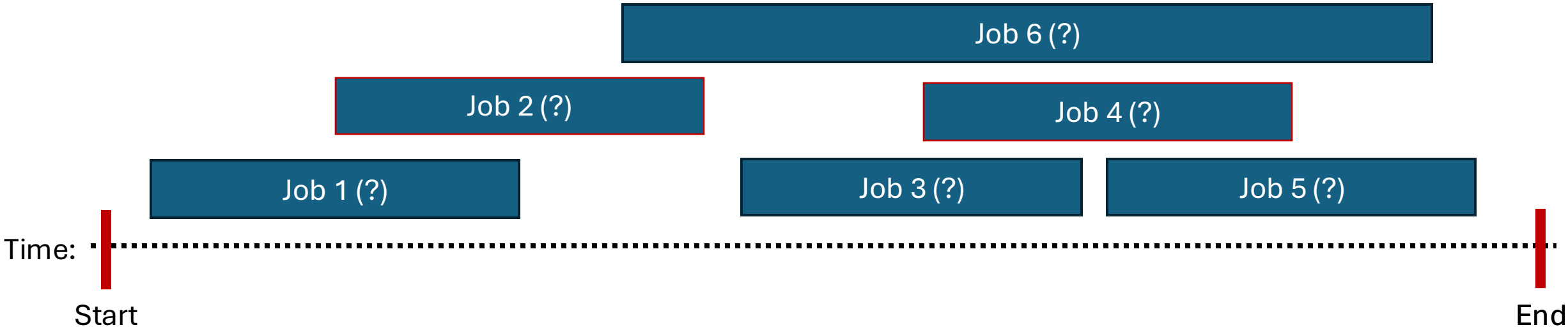
Weighted Interval Scheduling

- **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.



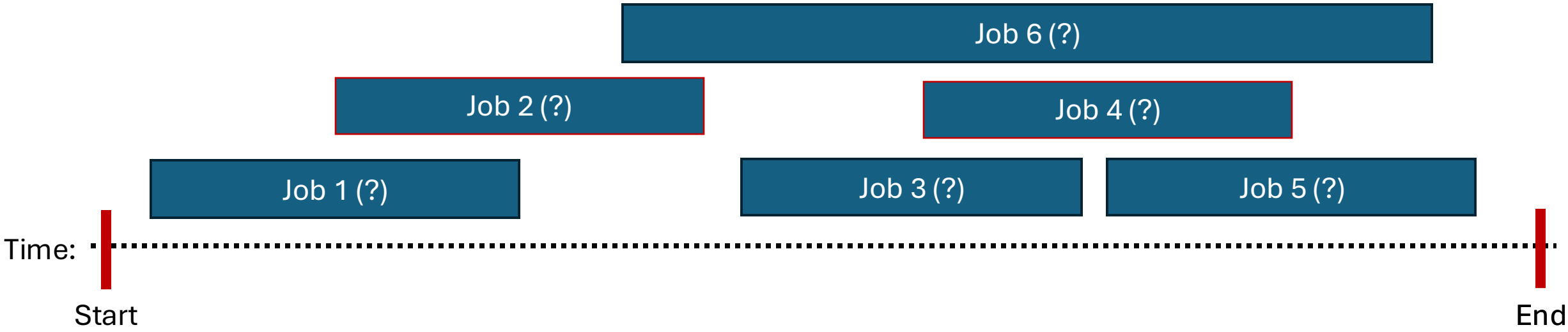
Divide & Conquer Approach

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



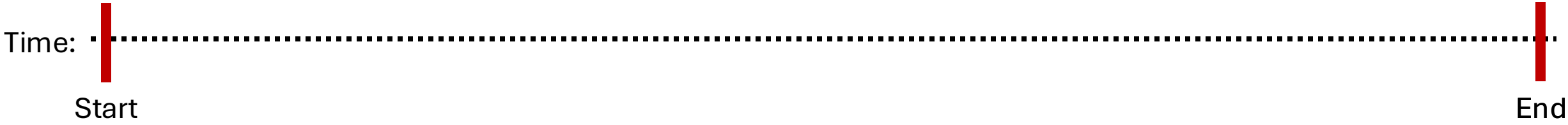
Binary Choice

- **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

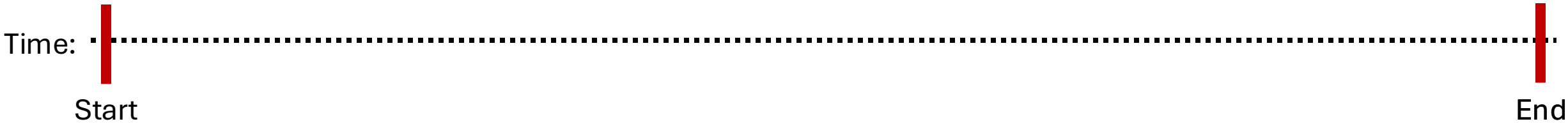
- Assume our list of n jobs are sorted by finish times.
- For all $j \in [n]$,
  - Let S_j be the optimal solution on the first j jobs.
  - Let OPT(j) be the value of that solution.
  - Let p(j) be largest i such that $i < j$ and Job i is computable with Job j.
    - Let p(j) = 0 if no jobs exist.



Binary Choice

- Now we can write

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

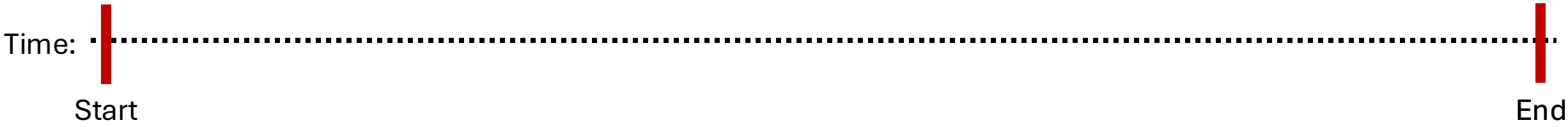


Binary Choice

- 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

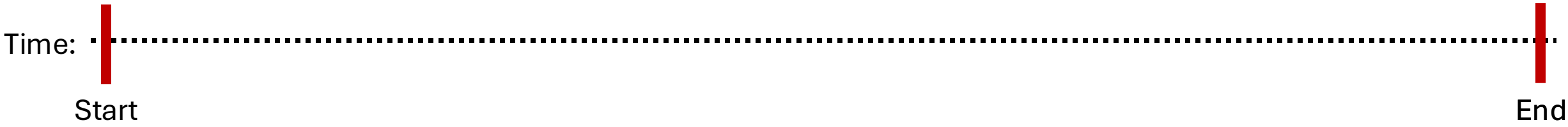
- Now we can write

We take Job J.

We consider the next job.

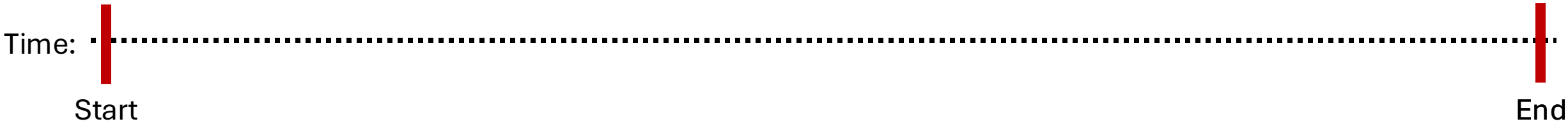
$$OPT(j) = max\{(v_j + OPT(p(j)), OPT(j-1)\}$$

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.



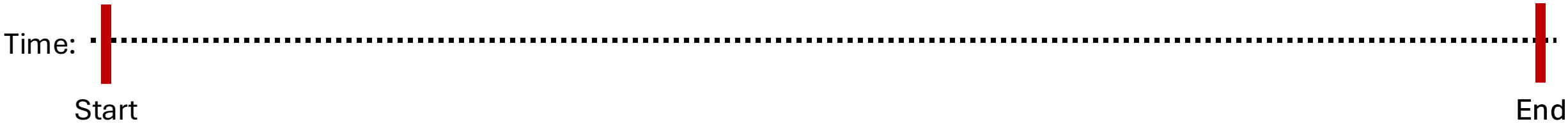
Binary Choice Algorithm

```
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?

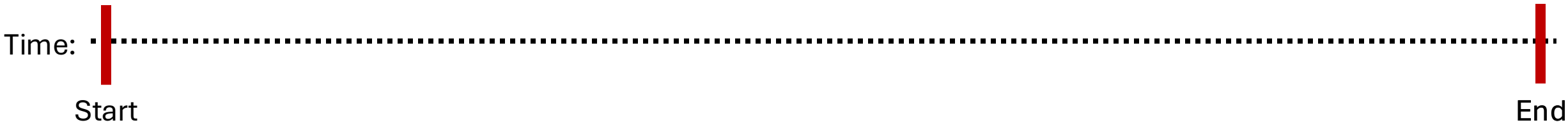
```
Brute-Force(L):
      Sort L by job finish times.
      Compute p[i] for each i using binary search.
      Return Compute-Opt(n)

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?

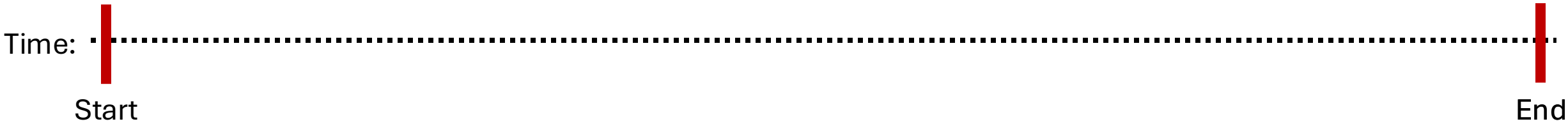
```
Brute-Force(L):
      Sort L by job finish times.
      Compute p[i] for each i using binary search.
      Return Compute-Opt(n)

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

O(nlog(n))



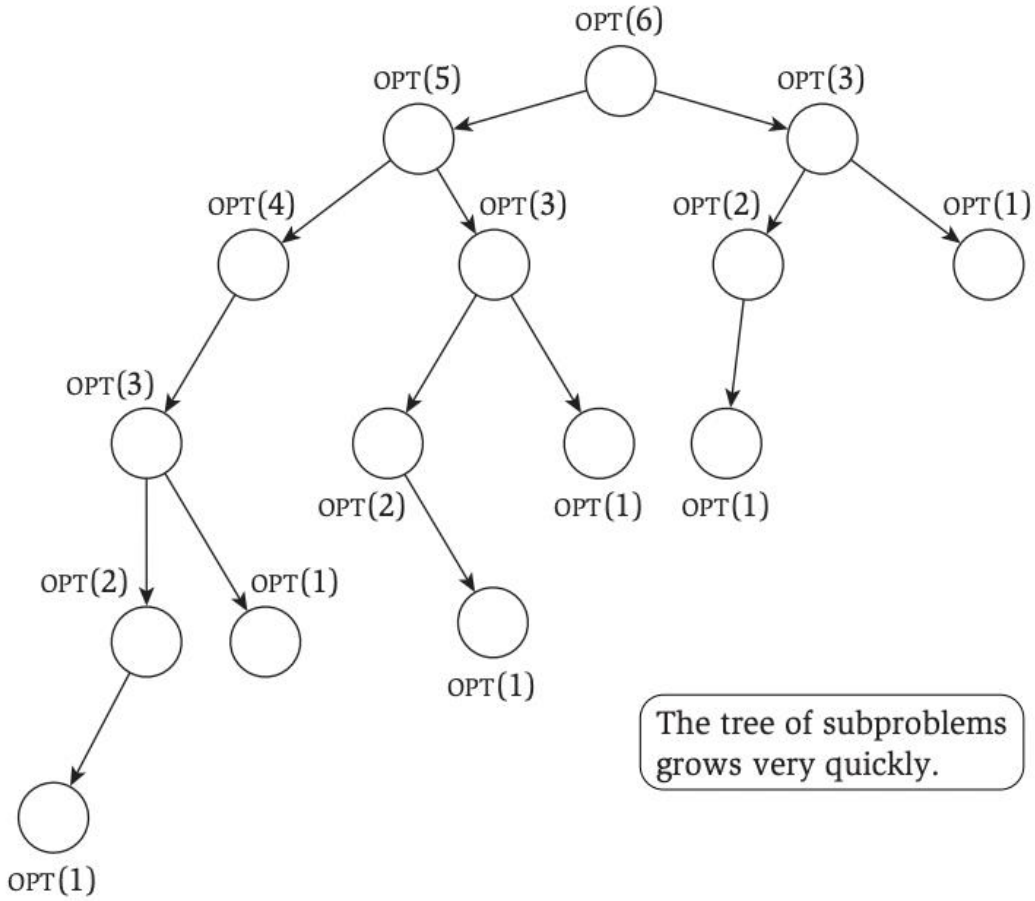
Answer: Could be exponential

```
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]))}
```



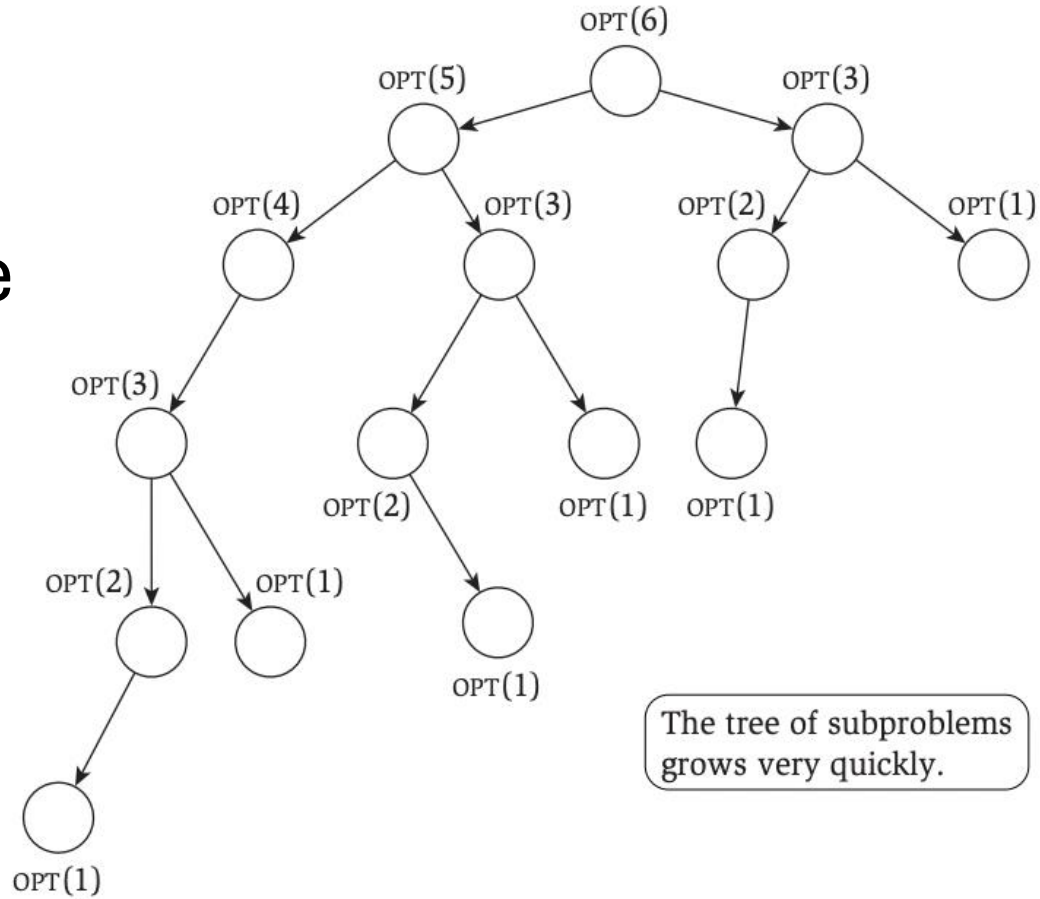
Recursion Tree

- 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!



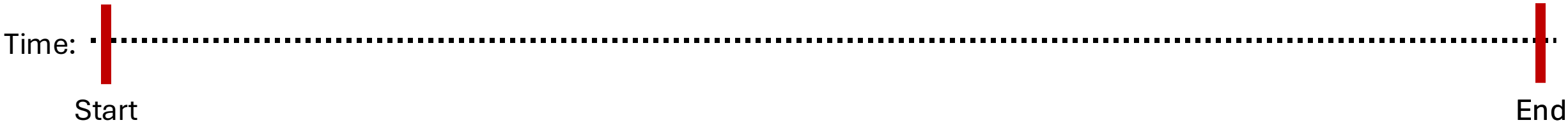
Memoization

- 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.



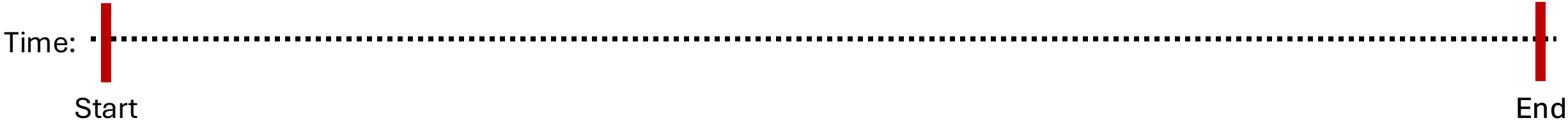
Memoization Runtime

```
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]
```



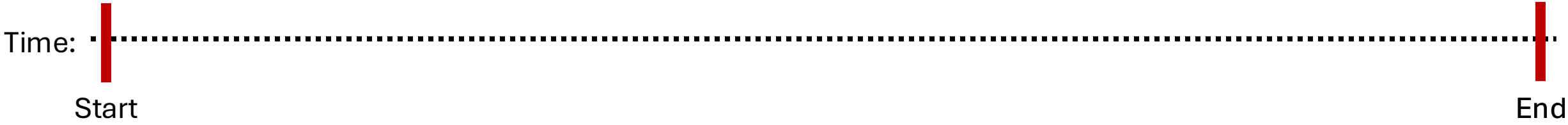
Memoization Runtime

- 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.



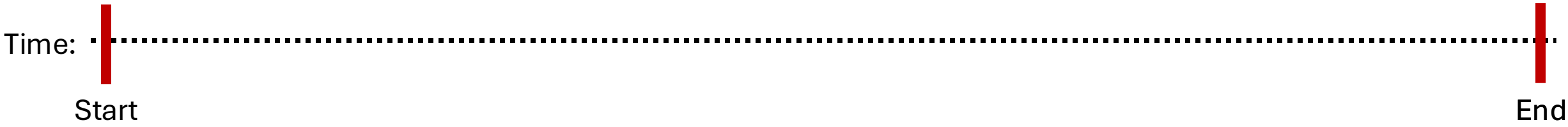
Memoization Runtime

- 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.

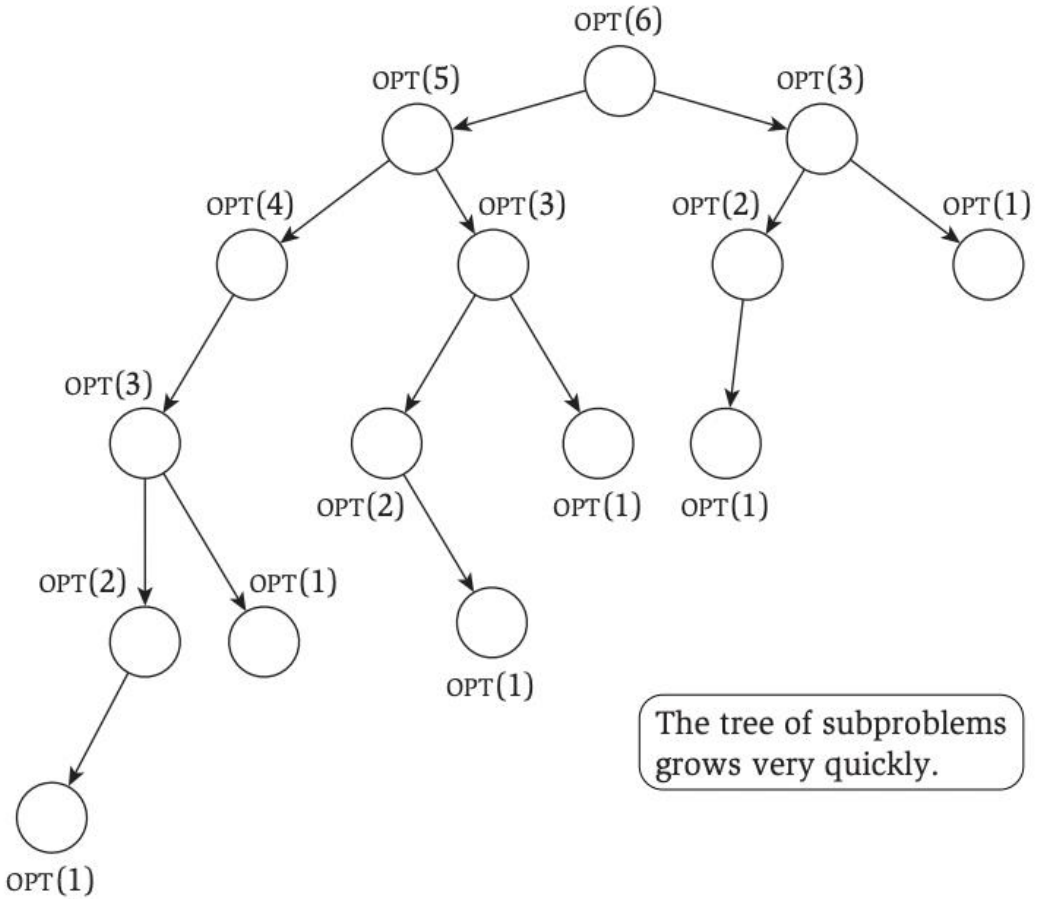


Top-Down Dynamic Programming

```
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



Recovering Solution

```
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]
```

Question: How do we recover the solution?

Recovering Solution

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

Find-Solution(j):
     If j == 0:
          return []
     Else if v[j] + M-Compute-Opt(p[j])} > M-Compute-Opt(j-1):
          return [j] ++ Find-Solution(p[j])
     Else:
          return Find-Solution(j-1)
```

Recovering Solution - O(n) Time

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

Find-Solution(j):
     If j == 0:
          return []
     Else if v[j] + M-Compute-Opt(p[j])} > M-Compute-Opt(j-1):
          return [j] ++ Find-Solution(p[j])
     Else:
          return Find-Solution(j-1)
```

Bottom-Up

```
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]
```

Question: What can we say about M[1]?

Bottom-Up

```
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]
```

Answer: To compute M[1], we only need M[0].

Bottom-Up

```
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]
```

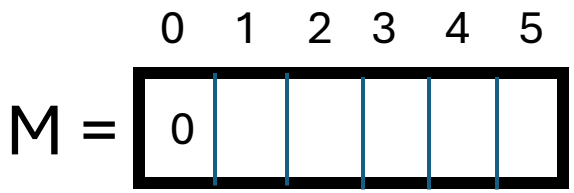
Question: What can we say about M[j]?

Bottom-Up

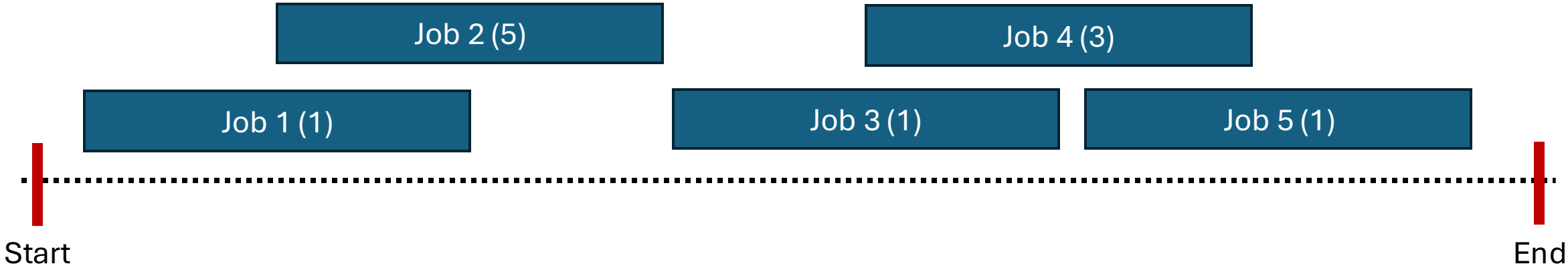
```
Brute-Force(L):
     Sort L by job finish times.
     Compute p[i] for each i using binary search.
     Global M = [], M[0] = 0
     Return Compute-Opt(n)

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]
```

Answer: To compute M[j], we only need M[i] for i < j.

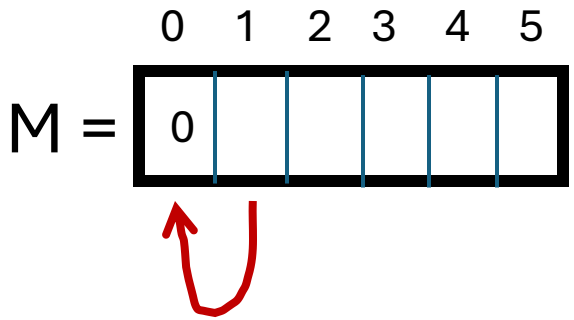
Bottom-Up



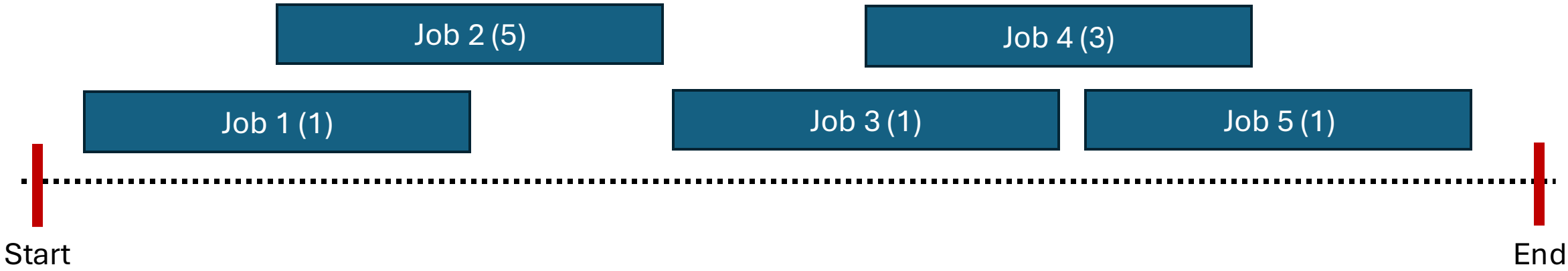

```
M-Compute-Opt(j):
      If j not in M:
            M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j])}
      Return M[j]
```



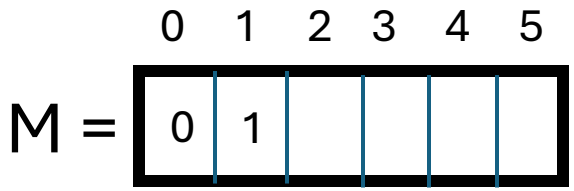
Bottom-Up



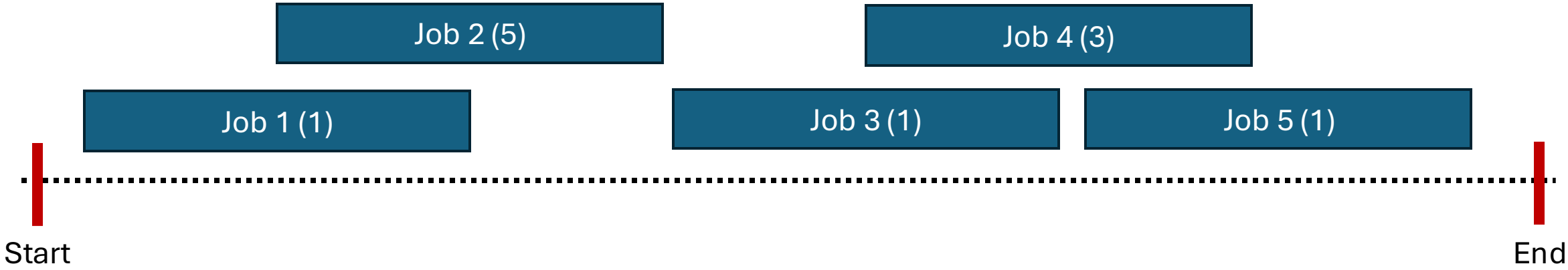

```
M-Compute-Opt(j):
        If j not in M:
                M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]))}
        Return M[j]
```



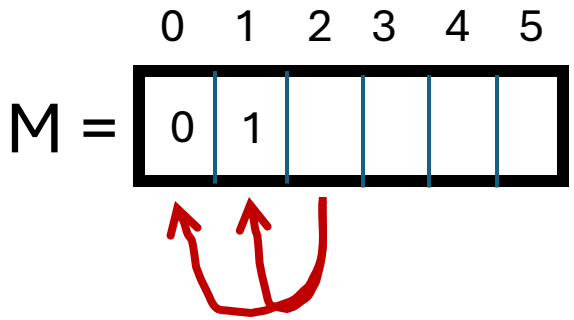
Bottom-Up



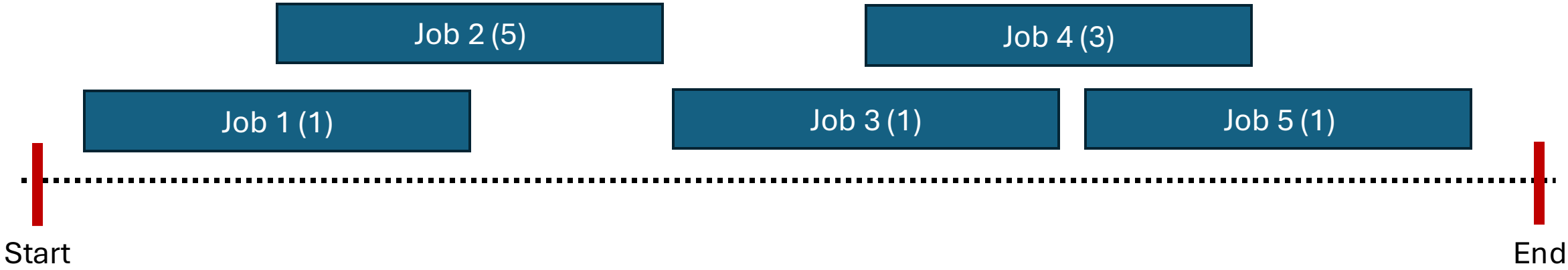

```
M-Compute-Opt(j):
      If j not in M:
            M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j])}
      Return M[j]
```



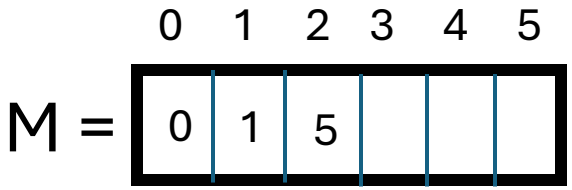
Bottom-Up



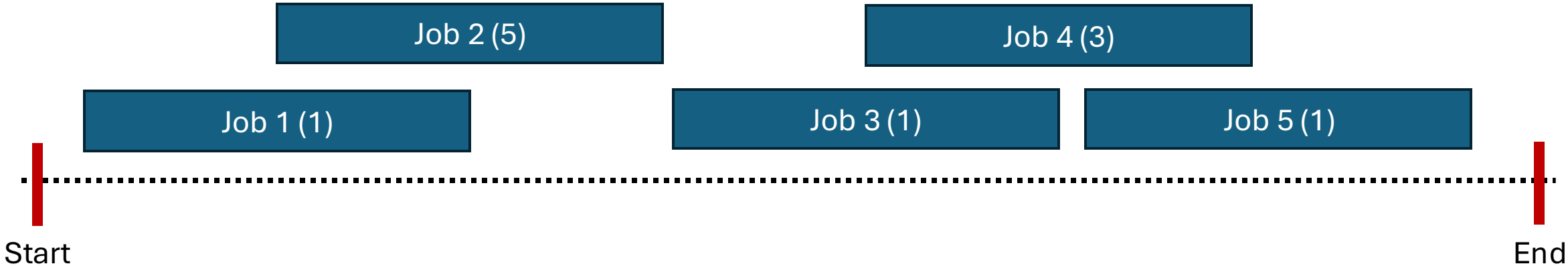

```
M-Compute-Opt(j):
        If j not in M:
                M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]))}
        Return M[j]
```



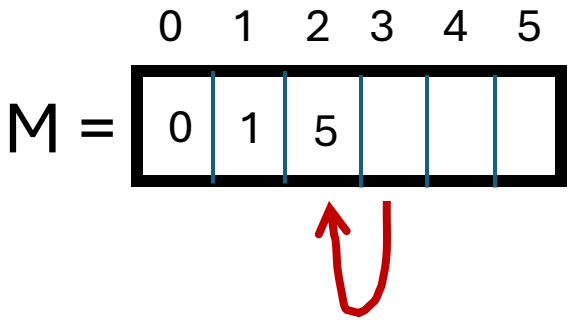
Bottom-Up



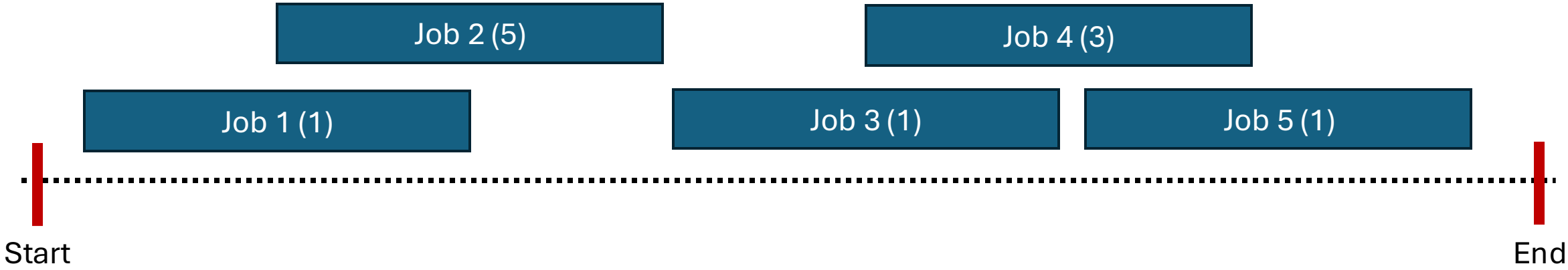

```
M-Compute-Opt(j):
      If j not in M:
            M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j])}
      Return M[j]
```



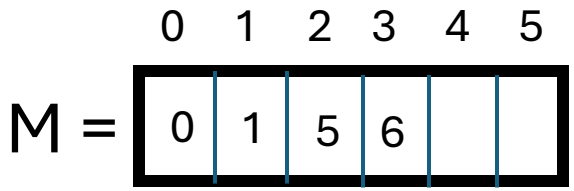
Bottom-Up



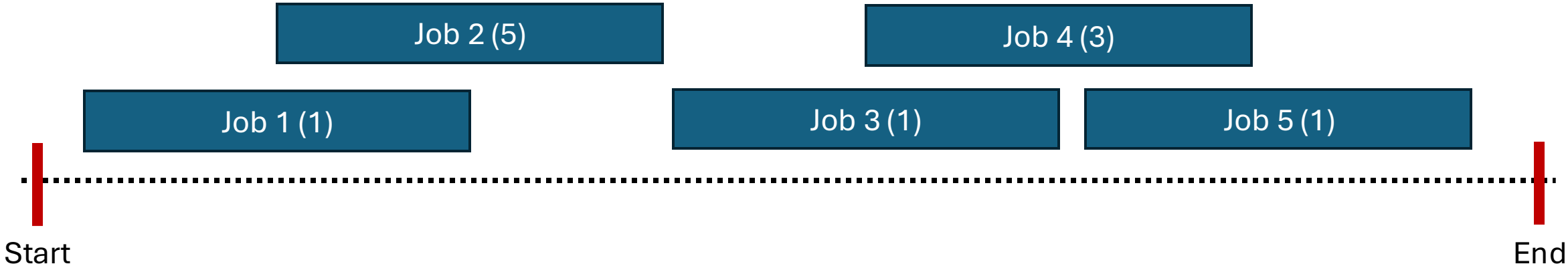

```
M-Compute-Opt(j):
        If j not in M:
                M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]))}
        Return M[j]
```



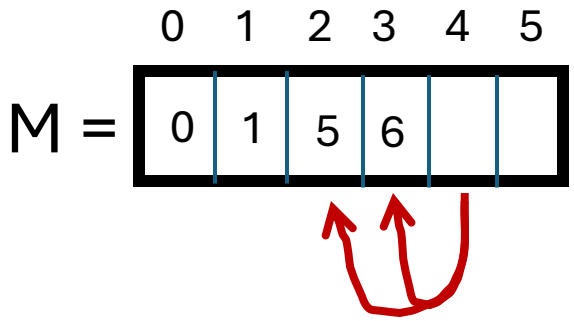
Bottom-Up



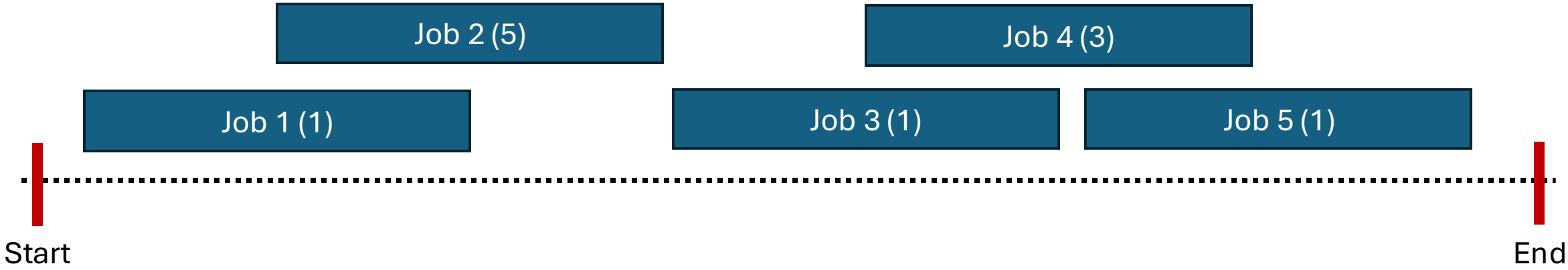

```
M-Compute-Opt(j):
      If j not in M:
            M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j])}
      Return M[j]
```



Bottom-Up



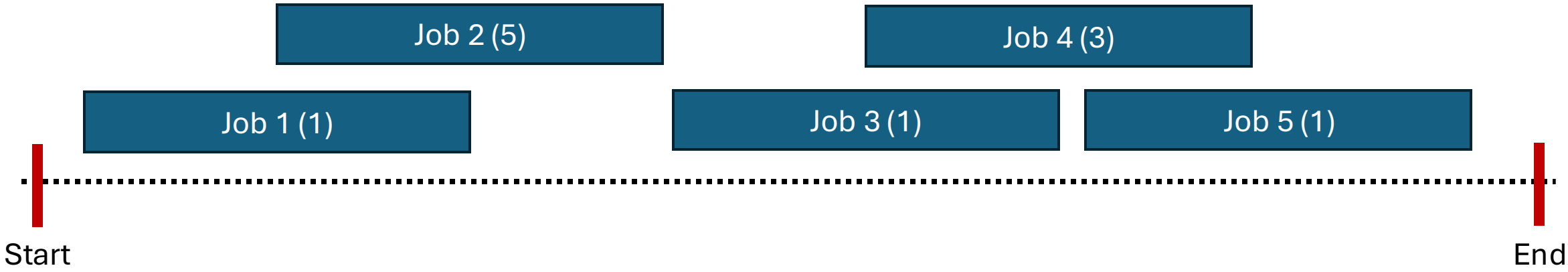

```
M-Compute-Opt(j):
        If j not in M:
                M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]))}
        Return M[j]
```



Bottom-Up

	0	1	2	3	4	5
M =	0	1	5	6	8	

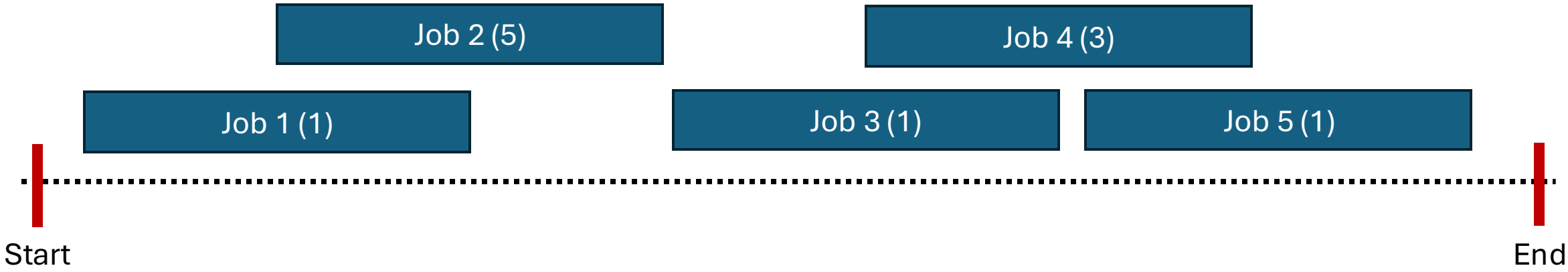
```
M-Compute-Opt(j):
      If j not in M:
            M[j] = Max(M-Compute-Opt(j-1), v[j] + M-Compute-Opt(p[j]))}
      Return M[j]
```



Bottom-Up

	0	1	2	3	4	5
M =	0	1	5	6	8	8

```
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]
```



Bottom-Up

```
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]
```

Instead of letting recursion decide the order, you compute them from the ”bottom” cases and work your way up:

```
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]
```