Intro to Dynamic Programming
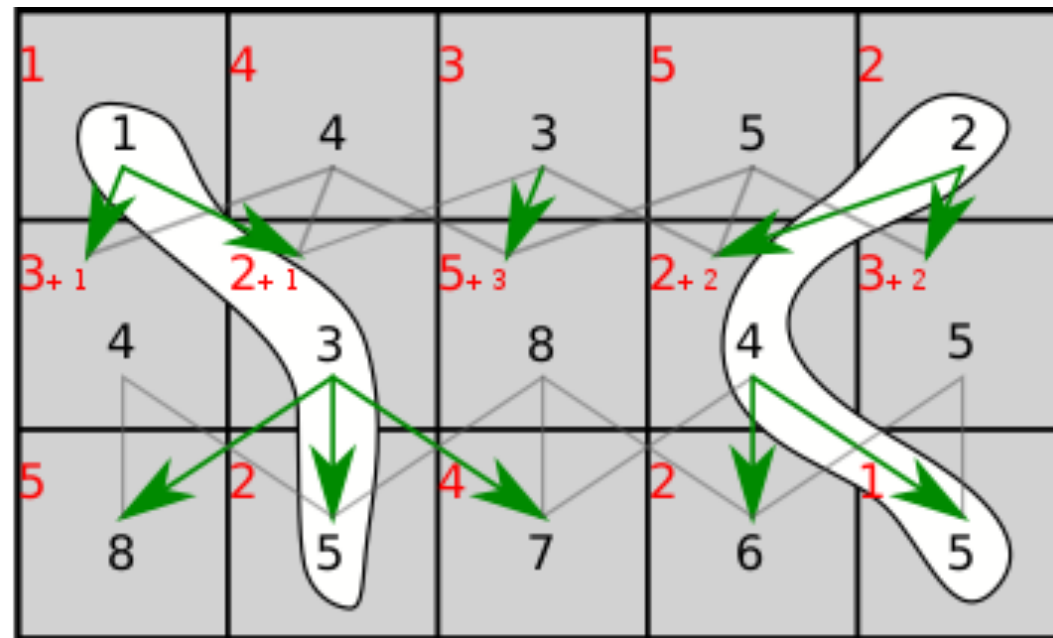
Admin

- Assignment 4 due tonight: **note on Q1(d)**

- **Midterm** next Friday **(April 2);** no class

  - Will be released 10.00 am Friday; can be taken in any 24 hour period between **10.00 am Friday to 10.00 am Sunday**

- Next assignment **HW 5** released today and is due a day early

  - Shorter HW: 3 problems

  - Due **Tues March 30 11 pm**

  - To give us enough time to grade and return feedback

  - Refrain from taking a late day otherwise might not be graded

CS Colloquium: 7 pm Today

"LOOK FOR THE HELPERS: CREATING AND MAINTAINING A CULTURE OF ADVOCACY/ACTIVISM IN COMPUTING+TECH

Dr. Nicki Washington is a professor of the practice of computer science at Duke University and the author of Unapologetically Dope: Lessons for Black Women and Girls on Surviving and Thriving in the Tech Field. Her career in higher education began at Howard University as the first Black female faculty member in the Department of Computer Science. Her professional experience also includes Winthrop University, The Aerospace Corporation, and IBM. She is a graduate of Johnson C. Smith University (B.S., '00) and North Carolina State University (M.S., '02; Ph.D., '05), becoming the first Black woman to earn a Ph.D. in computer science at the university and 2019 Computer Science Hall of Fame Inductee. She is a native of Durham, NC..

Dynamic Programming

"Those who cannot remember the past are condemned to repeat it."

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás

Slow Recursion: Fibonnacci

- So far we have seen recursion examples that are smart and lead to efficient solutions

- This is not always the case

- For example,

  - Recursive Fibonacci

Definition. Recall Fibonacci numbers are defined by the following recurrence

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

Slow Recursion: Fibonnacci

- This naive recurrence is horribly slow

- Let $T(n)$ denote the # of recursive calls

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

- Can we lower bound this?

$\underline{\text{RecFibo}(n)}:$
  if $n = 0$
      return 0
  else if $n = 1$
      return 1
  else
      return $\text{RecFibo}(n-1) + \text{RecFibo}(n-2)$

Slow Recursion: Fibonnacci

- Let's prove it's exponential; can we lower bound the running time using techniques we already have?

- $T(n) = T(n-1) + T(n-2) + \Theta(1)$

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

- Level i has cost 2^i.

- There are $n/2$ levels

- $T(n) = \Omega(2^{n/2})$

Slow Recursion: Fibonnacci

- A more careful analysis shows

- $T(n) \geq F_n$ for all $n \geq 1$

- $F_n \geq \phi^{n-2} \approx 1.6^{n-2}$ where $\phi = \left(\dfrac{1 + \sqrt{5}}{2} \right)$ (Golden ratio)

$$
\begin{aligned}
&\underline{\text{RecFibo}(n):} \\
&\quad \text{if } n = 0 \\
&\qquad \text{return } 0 \\
&\quad \text{else if } n = 1 \\
&\qquad \text{return } 1 \\
&\quad \text{else} \\
&\qquad \text{return RecFibo}(n-1) + \text{RecFibo}(n-2)
\end{aligned}
$$

Memo(r)ization

- Recursive Fibonacci algorithm is slow because it computes the same functions over and over

- Can speed it up considerably by writing down the results of our recursive calls, and looking them up when we need them later
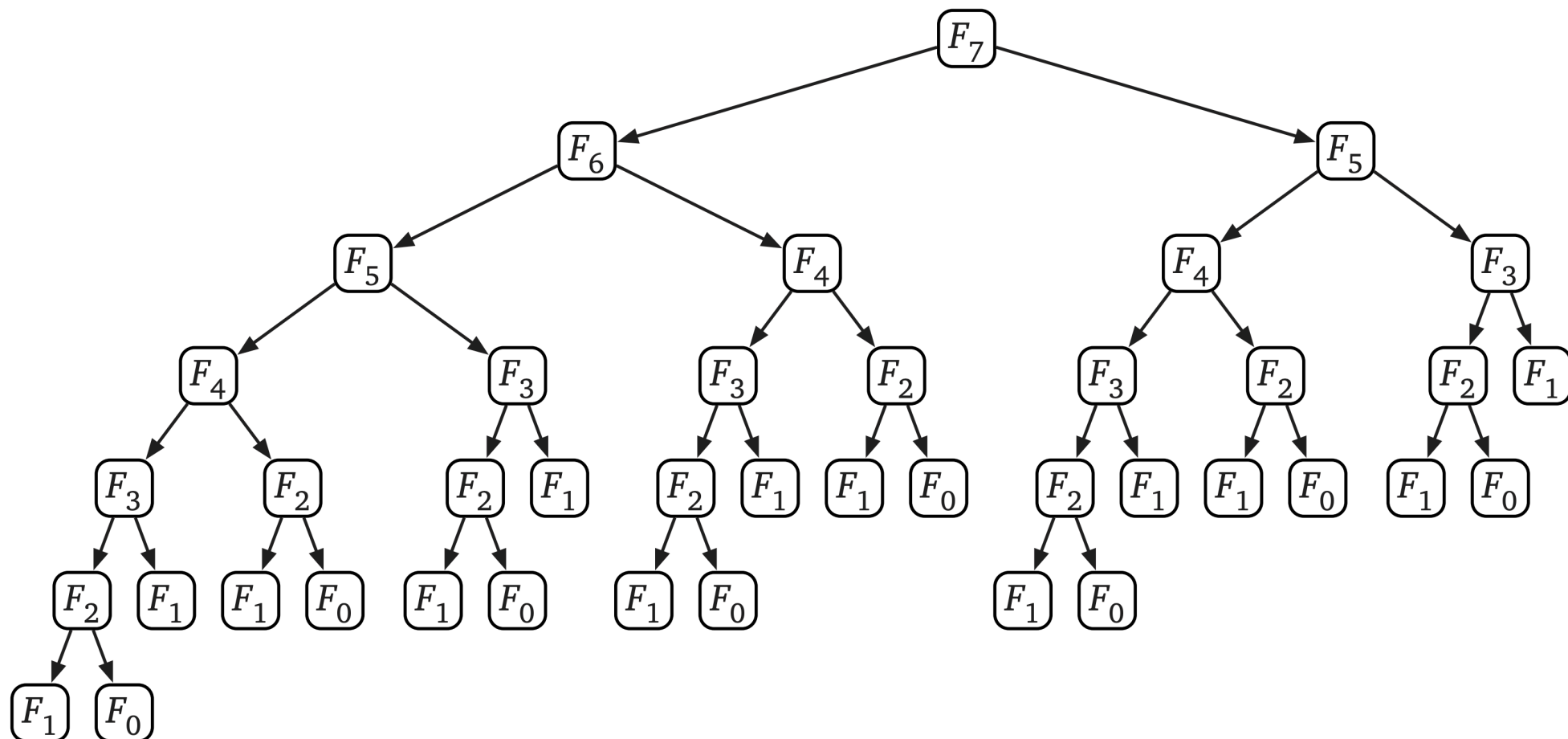
Dynamic Programming: Smart Recursion

- Dynamic programming is all about smart recursion by using memoization

- Here it cuts down on all useless recursive calls

$$F[n] = F[n-1] + F[n-2]$$
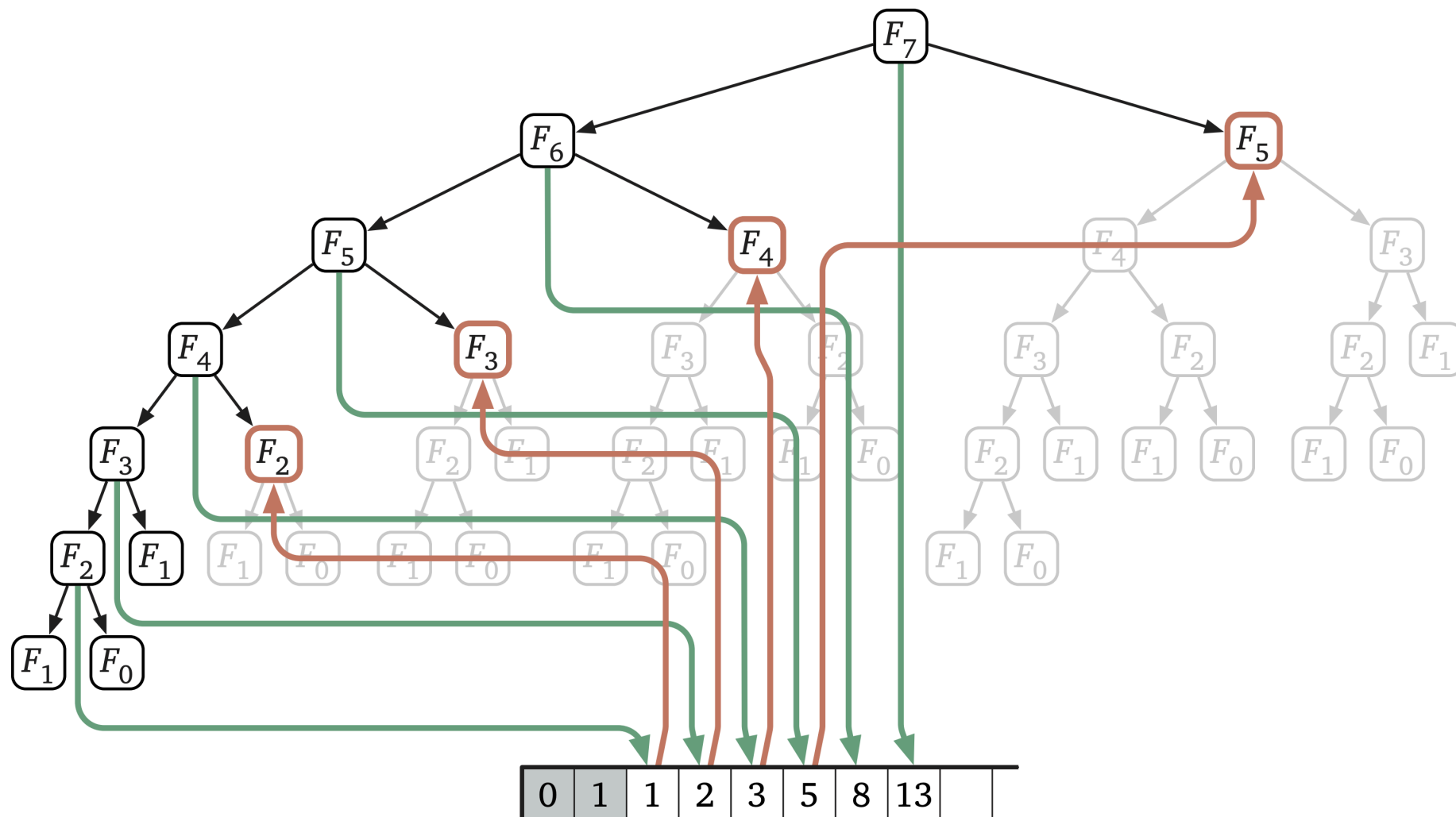
Dynamic Programing: Recursion + Memoization

- Memoization: technique to store expensive function calls so that they can be looked up later

- (Avoids calling the expensive function multiple times)

- A core concept of dynamic programming, but also used elsewhere

Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure

- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem

- **State the base case(s).** The subproblem thats so small we know the answer to it!

- **State the final answer.** (In terms of the subproblem)

- **Choose a memoization data structure.** Where are you going to store already computed results? (Usually a table)

- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which ones? Using these dependencies, identify an evaluation order

- **Analyze space and running time.** As always!

Weighted Scheduling

Reading: Chapter 6, KT

Weighted Scheduling

Job scheduling. Suppose you have a machine that can run one job at a time; n job requests, where each job i has a start time s_i, finish time f_i and weight $v_i \geq 0$.

Each job has a value

jobs d and g are incompatible
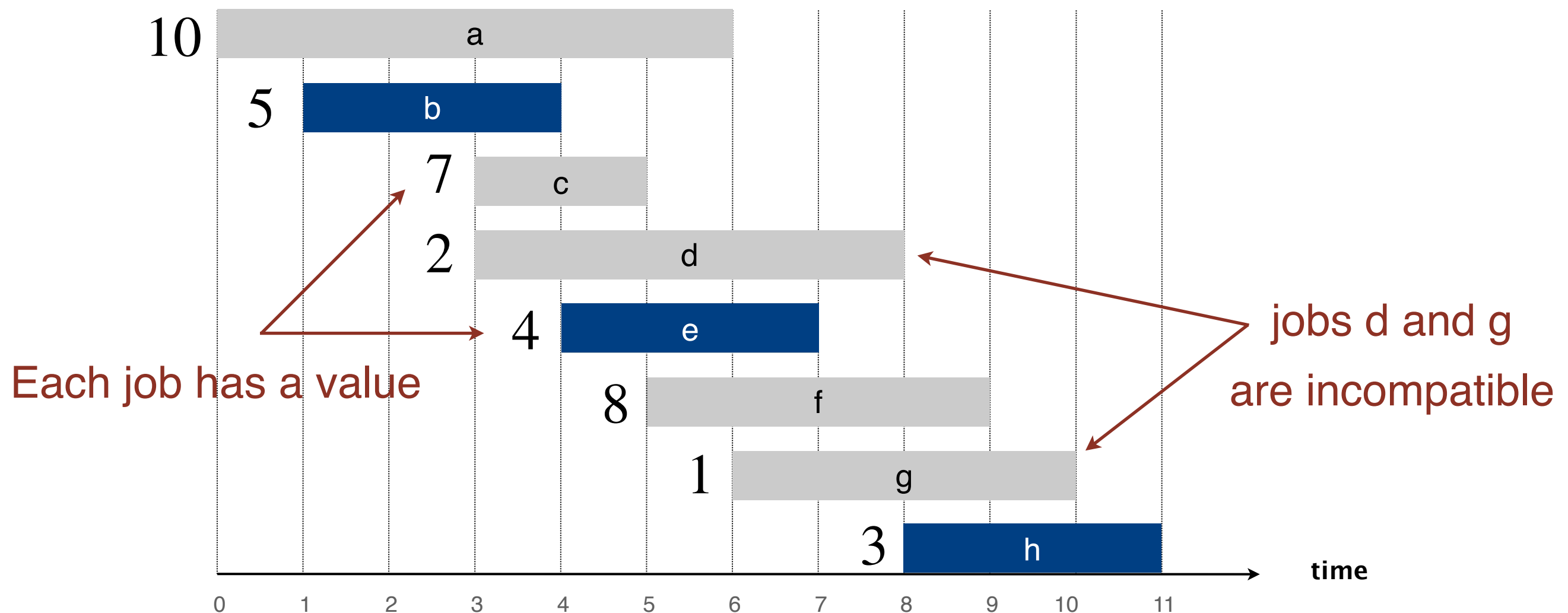
Weighted Scheduling

- **Input**. Given n intervals labeled $1, \ldots, n$ with starting and finishing times $(s_1, f_1), \ldots, (s_n, f_n)$ and each interval has a non-negative value or weight v_i

- **Goal**. We must select non-overlapping (compatible) intervals with the maximum weight. That is, our goal is to find $I \subseteq \{1, \ldots, n\}$ that are pairwise non-overlapping that maximize $\displaystyle\sum_{i \in I} v_i$

Remember Greedy?

- Greedy algorithm earliest-finish-time first

  - Considers jobs in order of finish times

  - Greedily picks jobs that are non-overlapping

- We proved greedy is optimal when all weights are one

- How about the weighted interval scheduling problem?
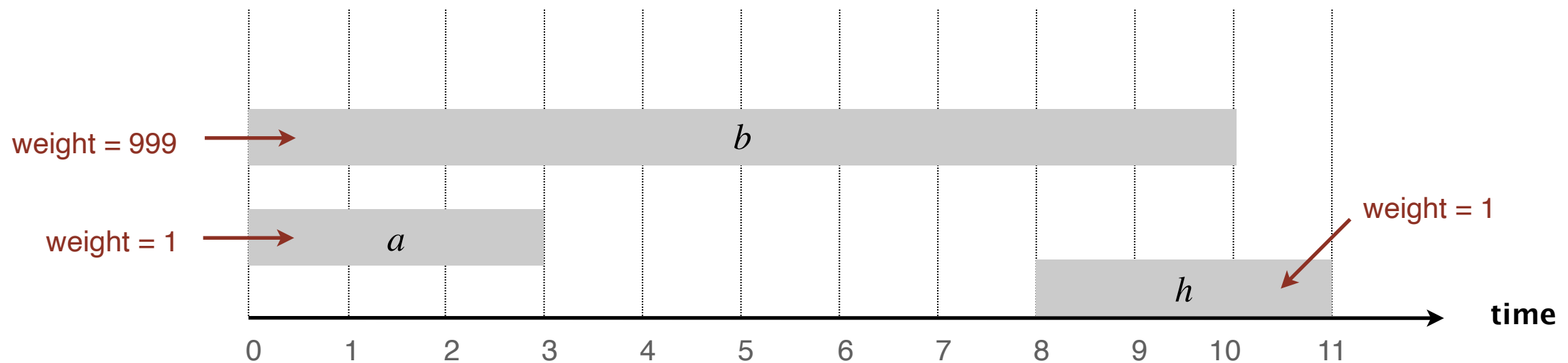
Greedy fails spectacularly

Different Greedy?

- A different greedy algorithm: greedily select intervals with the maximum weights, remove overlapping intervals

- Does that work?

weight = 10

weight = 9 each
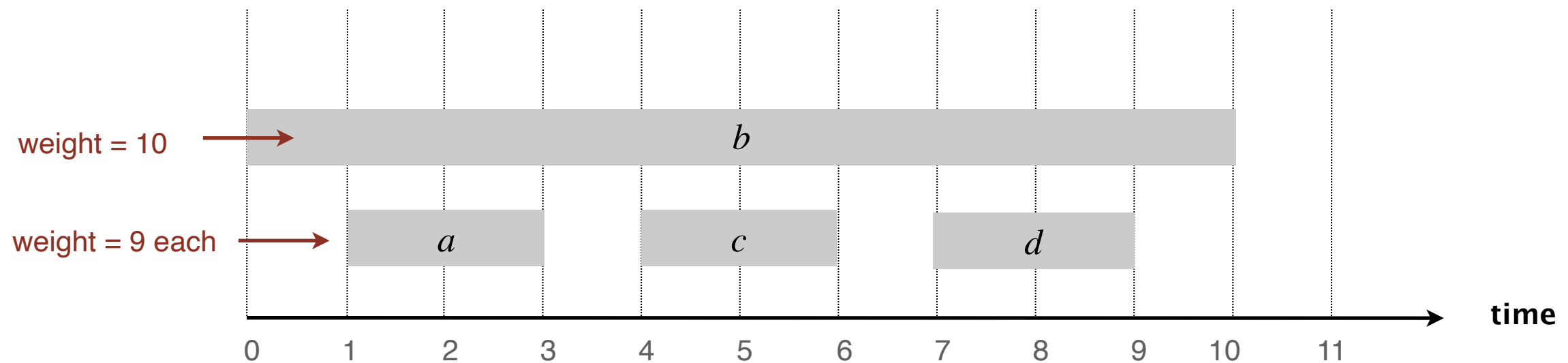
Greedy fails spectacularly

Let's Think Recursively

- The heart of dynamic programming is recursively thinking

- Coming up with a **smaller subproblem** which has the same optimal structure as the original problem

- First, to make things easy, we will focus on the total value of the optimal solution, rather than the actual optimal set, that is,

- **Optimal value.**
  Find the largest $\sum\limits_{i \in I} v_i$ where intervals in I are compatible.

- Opt-Schedule(n): the value of the optimal schedule of n intervals

Let's Think Recursively

- Consider the last interval: either it is in the optimal solution or not

- Whatever the overall optimal solution is, we can find it by considering both cases and taking the maximum over them

- **Case 1.** Last interval is not in the optimal solution

  - Remove it, we now have a smaller subproblem!

- **Case 2.** Last interval is in the optimal solution

  - Means anything overlapping with this interval cannot be in the solution, remove them

  - We have a smaller subproblem!

Formalize the Subproblem

Opt-Schedule(i): value of the optimal schedule that only uses intervals $\{1, \ldots, i\}$, for $0 \le i \le n$

Base Case & Final Answer

Opt-Schedule(i): value of the optimal schedule that only uses intervals $\{1,\ldots,i\}$, for $0 \leq i \leq n$

Base Case. Opt-Schedule$(0) = 0$

Goal (Final answer.) Opt-Schedule(n)

Recurrence
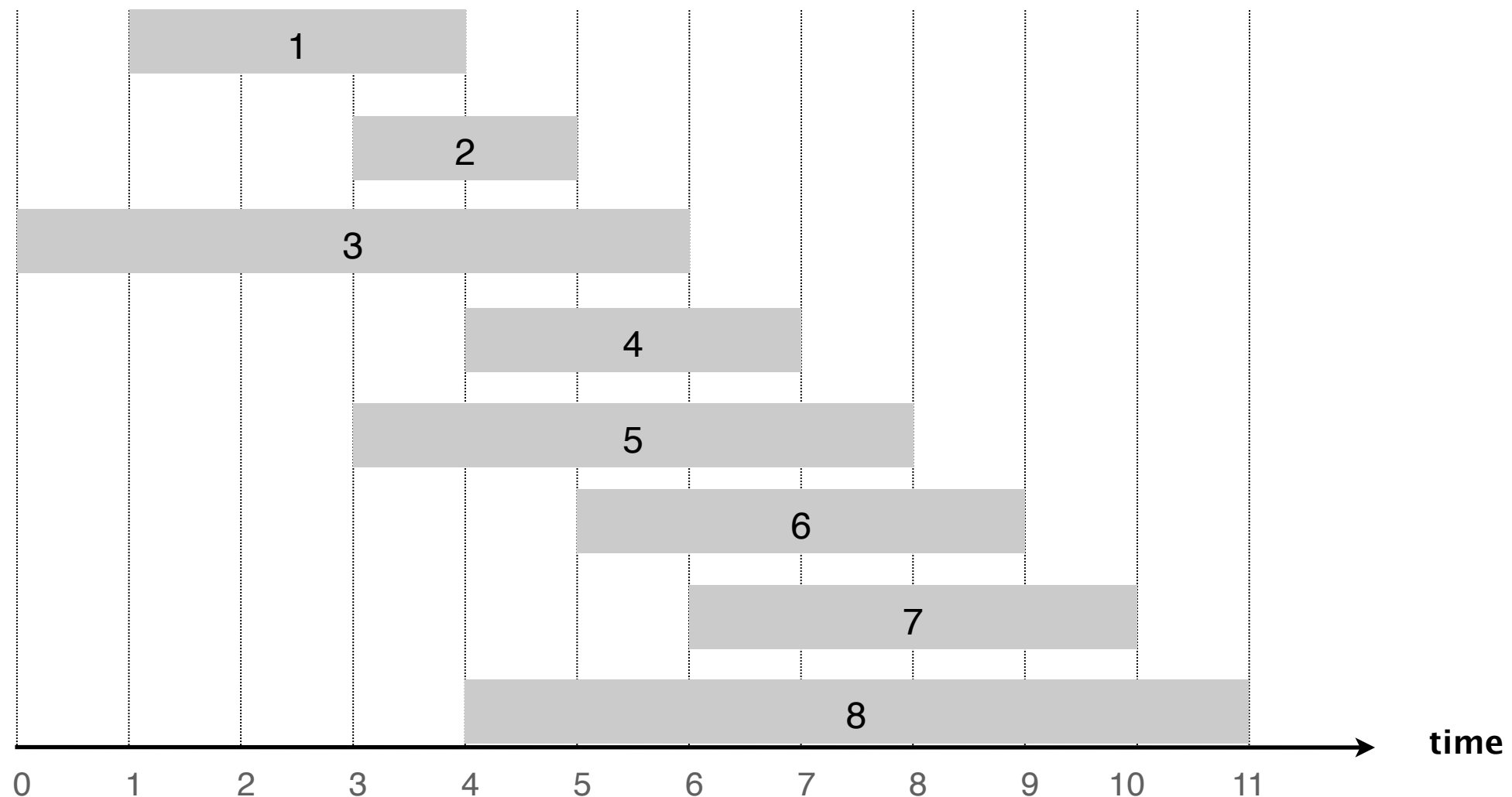
- How do we go from one subproblem to the next?

- The recurrence says how we can compute Opt-Schedule(i) by using values of Opt-Schedule(j) where $j < i$

- **Case 1.** Say interval i is not in the optimal solution, can we write the recurrence for this case?

  - Opt-Schedule$(i) =$ Opt-Schedule$(i - 1)$

- **Case 2.** Say interval i **is** in the optimal solution, what is the smaller subproblem we should recurse on in this case?

Recurrence

- The recurrence says how we can compute Opt-Schedule(i) by using values of Opt-Schedule(j) where $j < i$

- **Case 1.** Say interval i is not in the optimal solution:

  - Opt-Schedule$(i) =$ Opt-Schedule$(i - 1)$

- **Case 2.** Say interval i **is** in the optimal solution:

  - No interval $j < i$ that overlaps with i can be in solution

  - Need to remove all such intervals to get our smaller subproblem
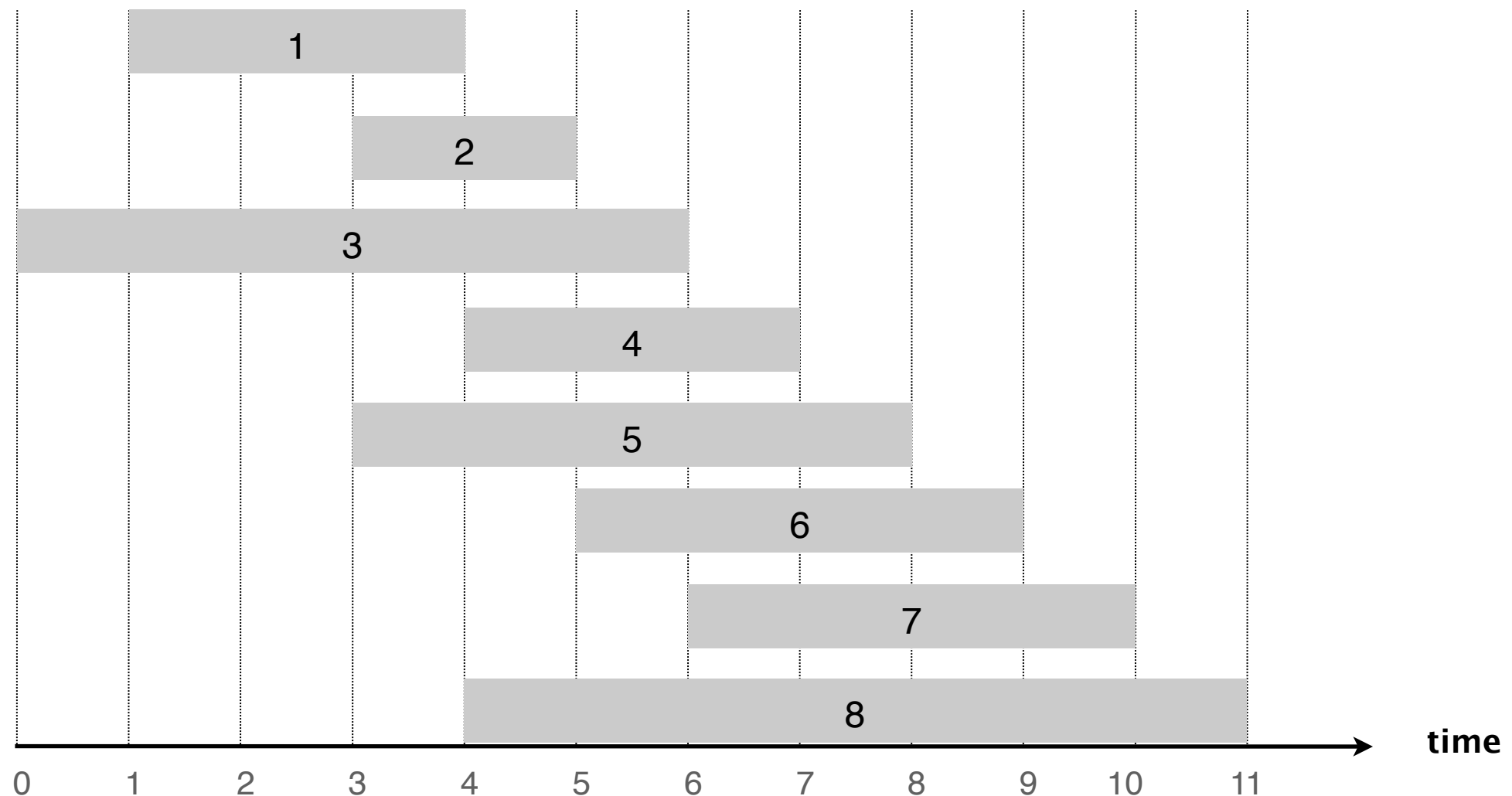
  - How do we do that?

Helpful Information

- Suppose the intervals are sorted by finish times

- Let $p(j)$ be the predecessors of j that is, largest index $i < j$ such that intervals i and j are not overlapping

- Define $p(j) = 0$ if all intervals $i < j$ overlap with j

Helpful Information

- Let $p(j)$ be the predecessors of j that is, largest index $i < j$ such that intervals i and j are not overlapping

  - $p(8) = ?, \quad p(7) = ?, \quad p(2) = ?$
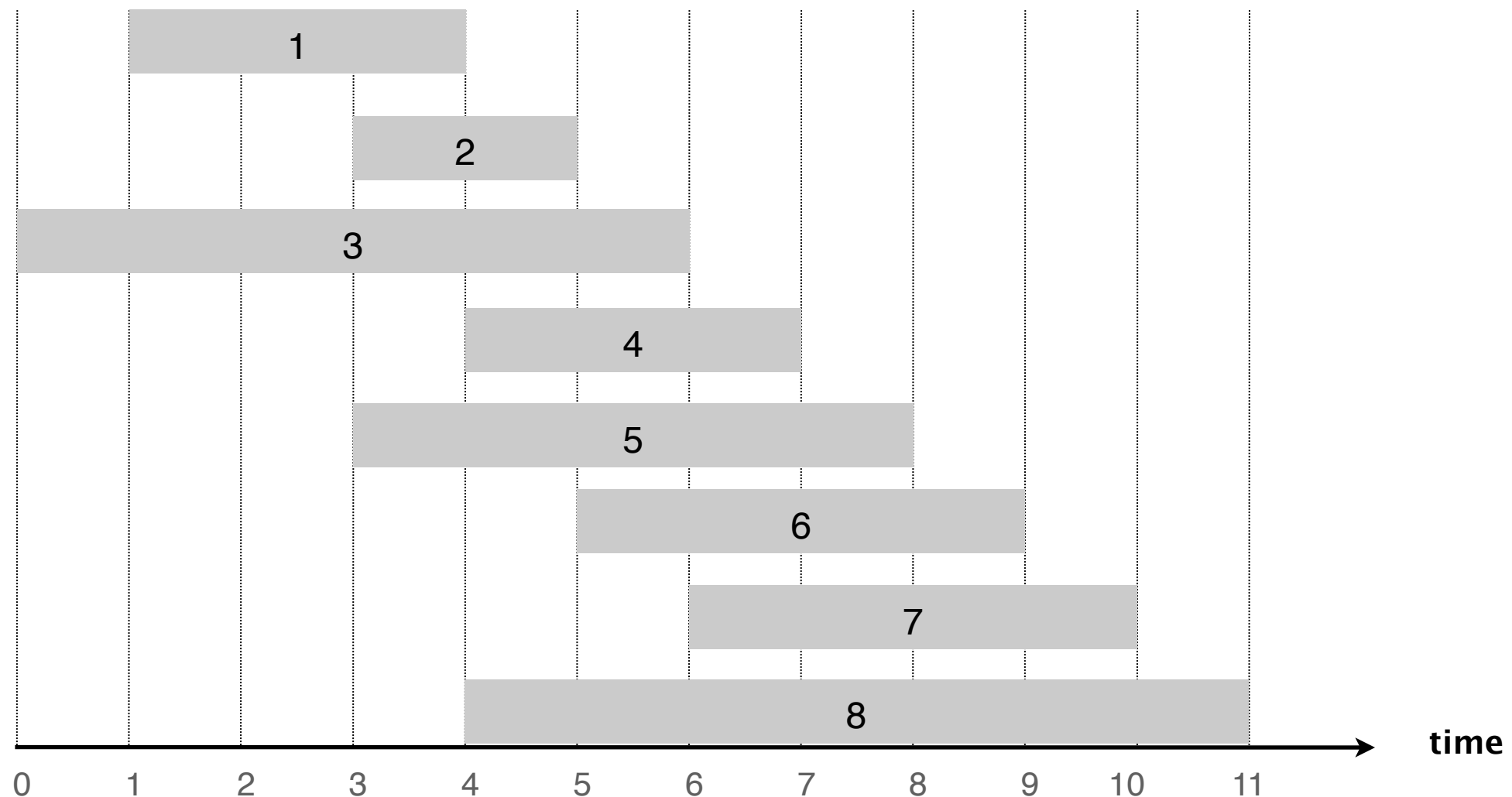
Helpful Information

- Let $p(j)$ be the predecessors of j that is, largest index $i < j$ such that intervals i and j are not overlapping

  - $p(8) = 1, \quad p(7) = 3, \quad p(2) = 0$

Recurrence

- The recurrence says how we can compute Opt-Schedule(i) by using values of Opt-Schedule(j) where $j < i$

- **Case 1.** Say interval i is not in the optimal solution:

  - Opt-Schedule$(i) =$ Opt-Schedule$(i - 1)$

- **Case 2.** Say interval i **is** in the optimal solution:

  - Suppose I know $p(i)$ predecessor of i, how can I write the recurrence for this case?

  - Opt-Schedule$(i) =$ Opt-Schedule$(p(i)) + v_i$

DP Recurrence

$$\text{Opt-Schedule}(i) =$$

$$\max\{\text{Opt-Schedule}(i-1), v_i + \text{Opt-Schedule}(p(i))\}$$

Filling Out the DP Table

0				
0	1	2	3	4
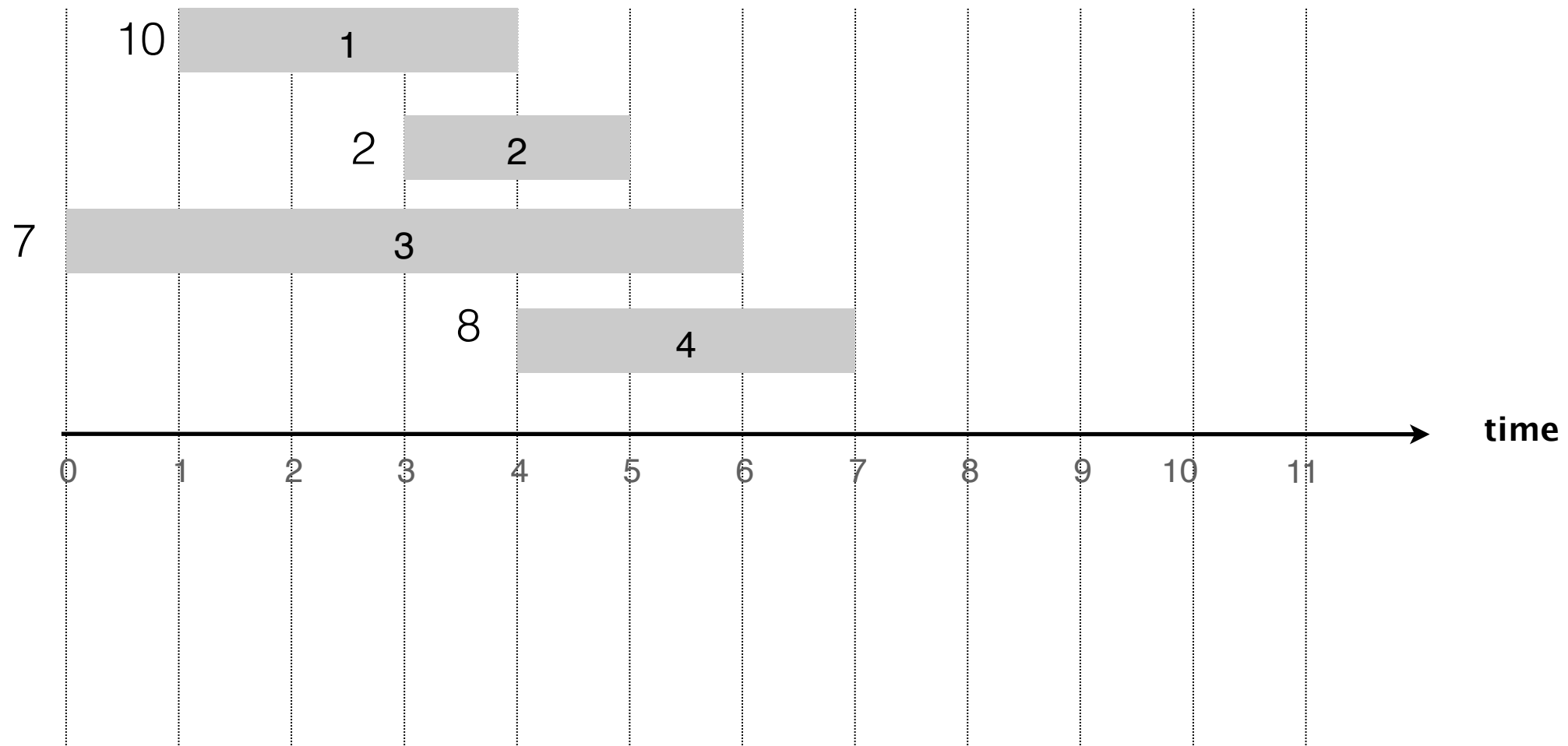
Filling Out the DP Table

0	10			
0	1	2	3	4
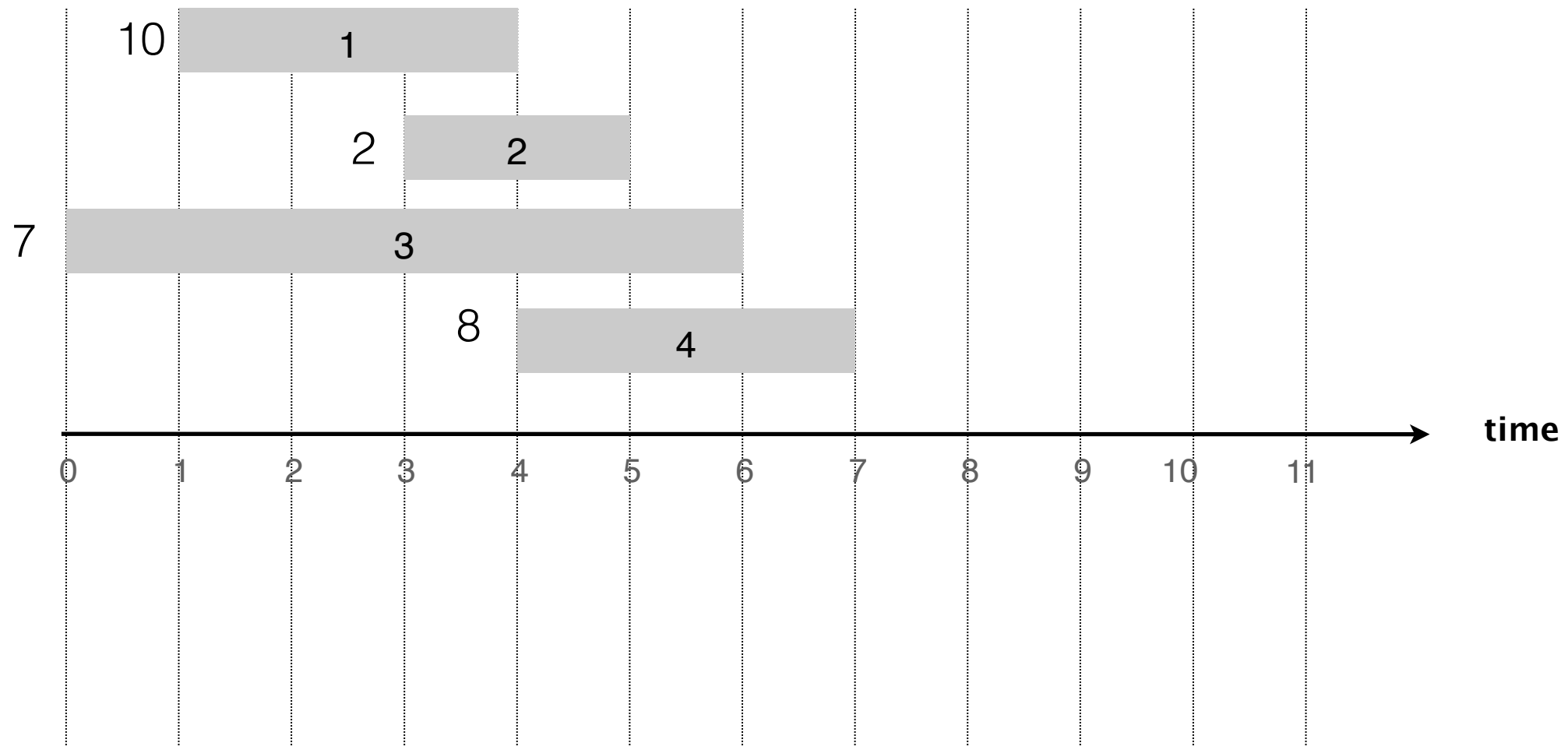
Filling Out the DP Table

0	10	10		
0	1	2	3	4
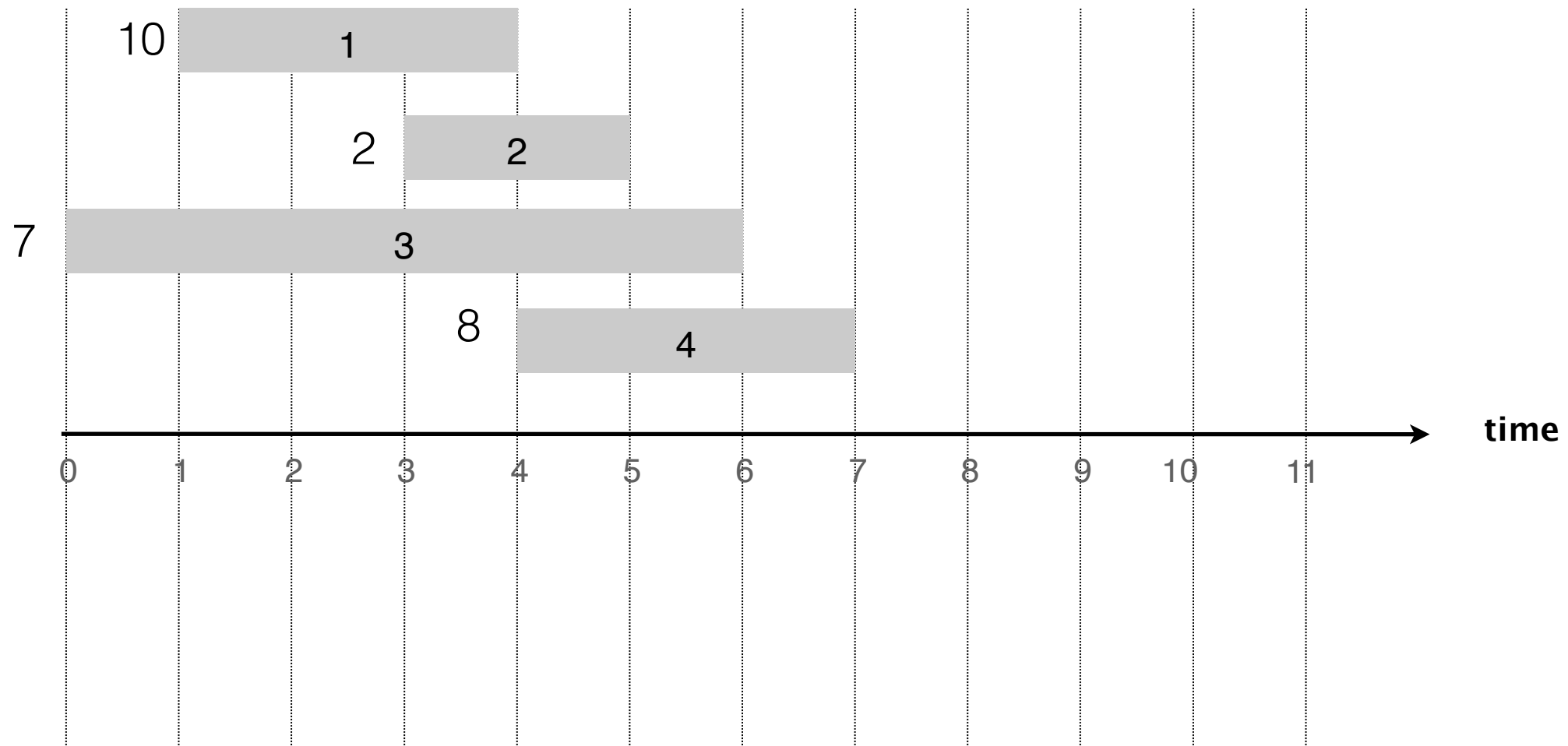
Filling Out the DP Table

0	10	10	10	18
0	1	2	3	4
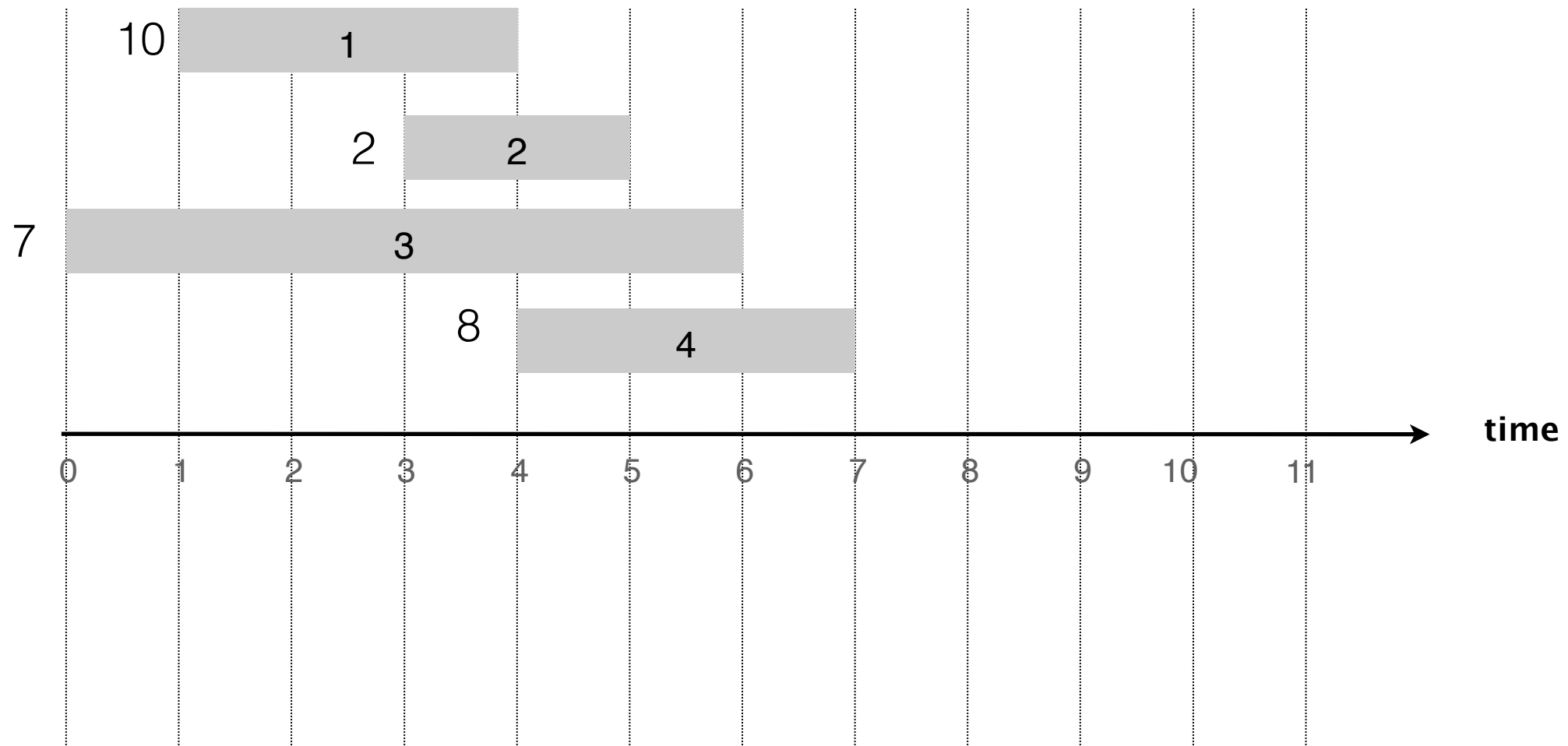
Summary of DP

- **Subproblem**.

  - For $0 \leq i \leq n,$ let Opt-Schedule(i) be the value of the optimal schedule that only uses intervals $\{1,\ldots,i\}$

  - Notice the optimal substructure

- **Recurrence.** Going from one subproblem to the next

  - Opt-Schedule$(i) =$
    $\max\{$Opt-Schedule$(i - 1),\ v_i +$ Opt-Schedule$(p(i))\}$

- **Base case**.

  - Opt-Scheduler$(0) = 0$ (no intervals to schedule)

- **Correctness**.

  - Using induction based on the recurrence

Remaining Pieces

- Final answer in terms of subproblem?

  - Opt-Schedule$[n]$

- Evaluation order (in what order can be fill the DP table)

  - $i = 0 \rightarrow n$, start with base case and use that to fill the rest

- Memoization data structure: 1-D array

- Final piece:

  - Running time and space

  - Space: $O(n)$

  - Time: preprocessing + time to fill array

Computing $p[i]$

- How quickly can we compute $p[i]$?

  - Can do a linear scan for each i: $O(i)$ per interval

  - Would be $O(n^2)$ overall

- We have intervals sorted by their finish time $F[1,\ldots,n]$

  - Can we use this?

  - For each interval, we can binary search over $F[1,\ldots,n]$, to need to find the first $j < i$ such that $f_j \leq s_i$

  - $O(\log n)$ for each interval

- Time $O(n \log n)$ to compute the array $p[]$

Running Time

- How many subproblems do we need to solve?

  - $O(n)$

- How long does it take to solve a subproblem?

  - $O(1)$ to take the max

- Preprocessing time:

  - Need to sort; $O(n \log n)$

  - Need to find $p(i)$ for all each i: $O(n \log n)$

- Overall: $O(n \log n) + O(n) = O(n \log n)$

- Space: $O(n)$

Recreating Chosen Intervals

- Suppose we have $M[]$ of optimal solutions

- How can we reconstruct the optimal set of intervals?

- When should an interval be included in the optimal?

- Depending on which of the two cases results in max tells us whether or not interval i is include:

  - Opt-Schedule$(i) =$
    $\max\{$Opt-Schedule$(i - 1),\ v_i +$ Opt-Schedule$(p(i))\}$

This value is bigger: i not in OPT

This value is bigger: i is in OPT

Recursive Solution?

Suppose for now that we do not memoize: just a divide and conquer recursion approach to the problem.

Opt-Schedule(i):

- If $j = 0$, return 0

- Else

  - Return $\max(\text{Opt-Schedule}(j - 1), v_j + \text{Opt-Schedule}(p(j)))$

- How many recursive calls in the worst case?

  - Depends on $p(i)$

- Can we create a bad instance?

Recursive Solution: Exponential

- For this example, asymptotically how many recursive calls?

- Grows like the Fibonacci sequence (exponential):
$$T(n) = T(n-1) + T(n-2) + O(1)$$

- Lots of redundancy!

  - How many distinct subproblems are there to solve?

  - Opt-Schedule(i) for $1 \leq i \leq n+1$

p(1) = 0, p(j) = j−2

recursion tree
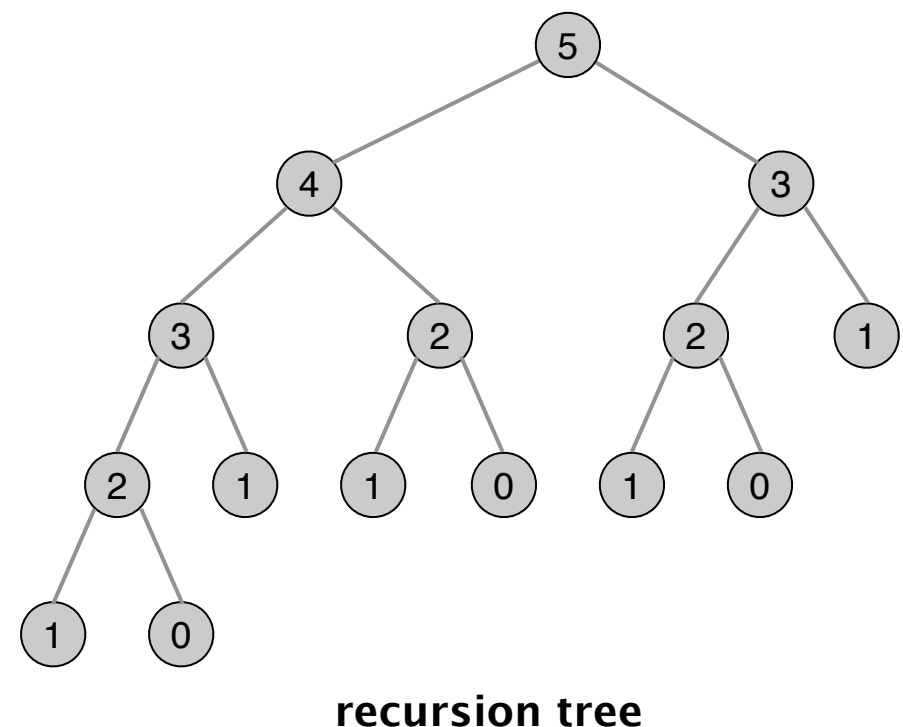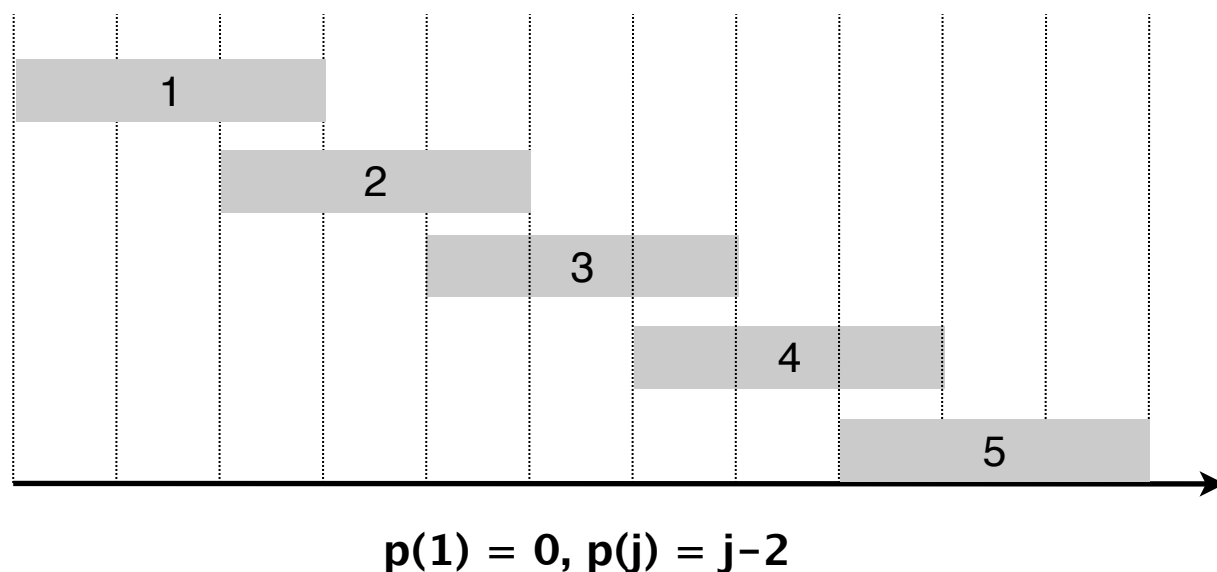
Dynamic Programming Tips

- Recurrence/subproblem is the key!

  - DP is a lot like divide and conquer, while writing extra things down

  - When coming to a new problem, ask yourself what subproblems may be useful? How can you break that subproblem into smaller subproblems?

  - Be clear while writing the subproblem and recurrence!

- In DP we usually keep track of the *cost* of a solution, rather than the solution itself

Longest Increasing Subsequence

Reading: Chapter 1, Erickson

Longest Increasing Subsequence

- Given a sequence of integers as an array $A[1,\ldots n]$, find the longest subsequence whose elements are in increasing order

- Find the longest possible sequence of indices
$1 \leq i_1 < i_2 < \ldots < i_\ell \leq n$ such that $A[i_k] < A[i_{k+1}]$

| **1** | **2** | 10 | **3** | 7 | 6 | **4** | **8** | **11** |

LIS: Length 6

| **1** | **2** | **10** | 3 | 7 | 6 | 4 | 8 | **11** |

A different increasing subsequence that is length 4

Longest Increasing Subsequence

- Given a sequence of integers as an array $A[1,\ldots n]$, find the longest subsequence whose elements are in increasing order

- Find the longest possible sequence of indices
  $1 \leq i_1 < i_2 < \ldots < i_\ell \leq n$ such that $A[i_k] < A[i_{k+1}]$

| **1** | **2** | 10 | **3** | 7 | 6 | **4** | **8** | **11** |

- Length of the longest increasing subsequence above is 6

- To simplify, we will only compute **length of the LIS**

Formalize the Subproblem

$L[i]$: length of the longest increasing subsequence in $A[1,\ldots,i]$ that ends at (and includes) $A[i]$

Identify the Base Case

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

Base Case. $L[1] = ?$

Identify the Final Answer

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

Base Case. $L[1] = 1$

Final answer. ?

Base Case & Final Answer

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

Base Case. $L[1] = 1$

Final answer. $\max_{1 \leq i \leq n} L[i]$

Recurrence

- How do we go from one subproblem to the next?

- That is, how do we compute $L[i]$ assuming I know the values of $L[1], \ldots, L[i-1]$

| **1** | **2** | 10 | **3** | 7 | 6 | **4** | **8** | **11** |

Length of the LIS ending at 2?

Length of the LIS ending at 10?

Recurrence

- Let's say we know the length of the longest subsequence ending at $A[1], A[2], \ldots A[i-1]$

- What is the longest subsequence ending at $A[i]$?

- $A[i]$ could potential extend an earlier subsequence:

  - Can extend a longest subsequence ending at some $A[k]$, with $A[k] < A[i]$, but which k?

  - OK, let's try all k to get the answer

- Or it doesn't extend any earlier increasing subsequence

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A | 1 | 2 | 10 | 3 | 7 | 6 | 4 | 8 | 11

L | 1 | | | | | | | |

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A

1	**2**	10	3	7	6	4	8	11

L

1	2							

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A

1	2	10	3	7	6	4	8	11

L

1	2	3						

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A

| **1** | **2** | 10 | **3** | 7 | 6 | 4 | 8 | 11 |

L

| 1 | 2 | 3 | 3 | | | | | |

How do we know 3 extends a past LIS?

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A

1	**2**	10	**3**	7	6	4	8	11

L

1	2	3	3					

$L[j]$ extends an LIS ending at $L[i]$ if $A[j] > A[i]$

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A | **1** | **2** | 10 | **3** | **7** | 6 | 4 | 8 | 11 |

L | 1 | 2 | 3 | 3 | 4 | | | | | |

$L[j]$ extends an LIS ending at $L[i]$ if $A[j] > A[i]$

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A

1	**2**	10	**3**	7	**6**	4	8	11

L

1	2	3	3	4	4				

$L[j]$ extends an LIS ending at $L[i]$ if $A[j] > A[i]$

Example: Building a Recurrence

$L[i]$: length of the longest increasing subsequence in A that ends at (and includes) $A[i]$

A | **1** | **2** | 10 | **3** | 7 | 6 | **4** | 8 | 11 |

L | 1 | 2 | 3 | 3 | 4 | 4 | 4 | | | |

$L[j]$ extends an LIS ending at $L[i]$ if $A[j] > A[i]$

LIS: Recurrence

$$L[j] = 1 + \max\{L[i] \mid i < j \text{ and } A[i] < A[j]\}$$

Assuming $\max \varnothing = 0$

Recursion → DP

- If we used recursion (without memoization) we'll be inefficient—we'll do a lot of repeated work

- Once you have your recurrence, the remaining pieces of the dynamic programming algorithm are

  - **Evaluation order.** In what order should I evaluate my subproblems so that everything I need is available to evaluate a new subproblem?

    - For LIS we just left-to-right on array indices

  - **Memoization structure.** Need a table (array or multi-dimensional array) to store computed values

    - For LIS, we just need a one dimensional array

    - For others, we may need a table (two-dimensional array)

LIS Analysis

- Correctness

  - Follows from the recurrence using induction

- Running time?

  - Solve $O(n)$ subproblems

  - Each one requires $O(n)$ time to take the min

  - $O(n^2)$

  - An Improved DP solution takes $O(n \log n)$

- Space?

  - $O(n)$ to store array $L[]$

Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure

- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem

- **State the base case(s)**. The subproblem thats so small we know the answer to it!

- **State the final answer.** (In terms of the subproblem)

- **Choose a memoization data structure.** Where are you going to store already computed results? (Usually a table)

- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which ones? Using these dependencies, identify an evaluation order

- **Analyze space and running time.** As always!

Dynamic Programming

- Formalized by Richard Bellman in the 1950s

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*". … 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?

- Chose the name **"dynamic programming"** to hide the mathematical nature of the work from military bosses

Acknowledgments

- Some of the material in these slides are taken from

  - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)

  - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)