Dynamic Programming

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

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás,

Admin

- Assignment 3 back
- Some common issues with greedy/exchange arguments
- “Greedy worksheet” linked to on course webpage
- We’ll go over greedy again in midterm review

Student solutions!

- PDF on **glow** containing student solutions to some assignment problems
- Idea: give you an idea of how non-polished proofs might look
- (No promises of perfect correctness, or ideal length, or everything being explained)
- Don’t distribute

Assignments

- Assignment 5 due Saturday
- Make sure to keep up with assignments!
- Midterm soon
- Weighted heavily
- Much better to let me know beforehand if there's an issue

Midterm review

- Next Monday evening
- (Any questions/comments about that time?)
- All remote
- I'll send around a survey about when people can make it
- You can send in questions if you can't attend
- Will post a recording

Apply to be a TA!

- Application form on dept website
- Fun, educational
- OK if remote (need to live in US)

Apply to be a TA!

- Application form on dept website
- Fun, educational
- OK if remote (need to live in US)

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?

```
RecFibo(n):
    if n = 0
        return 0
    else if n = 1
        return 1
    else
        return RecFibo(n − 1) + RecFibo(n − 2)
```

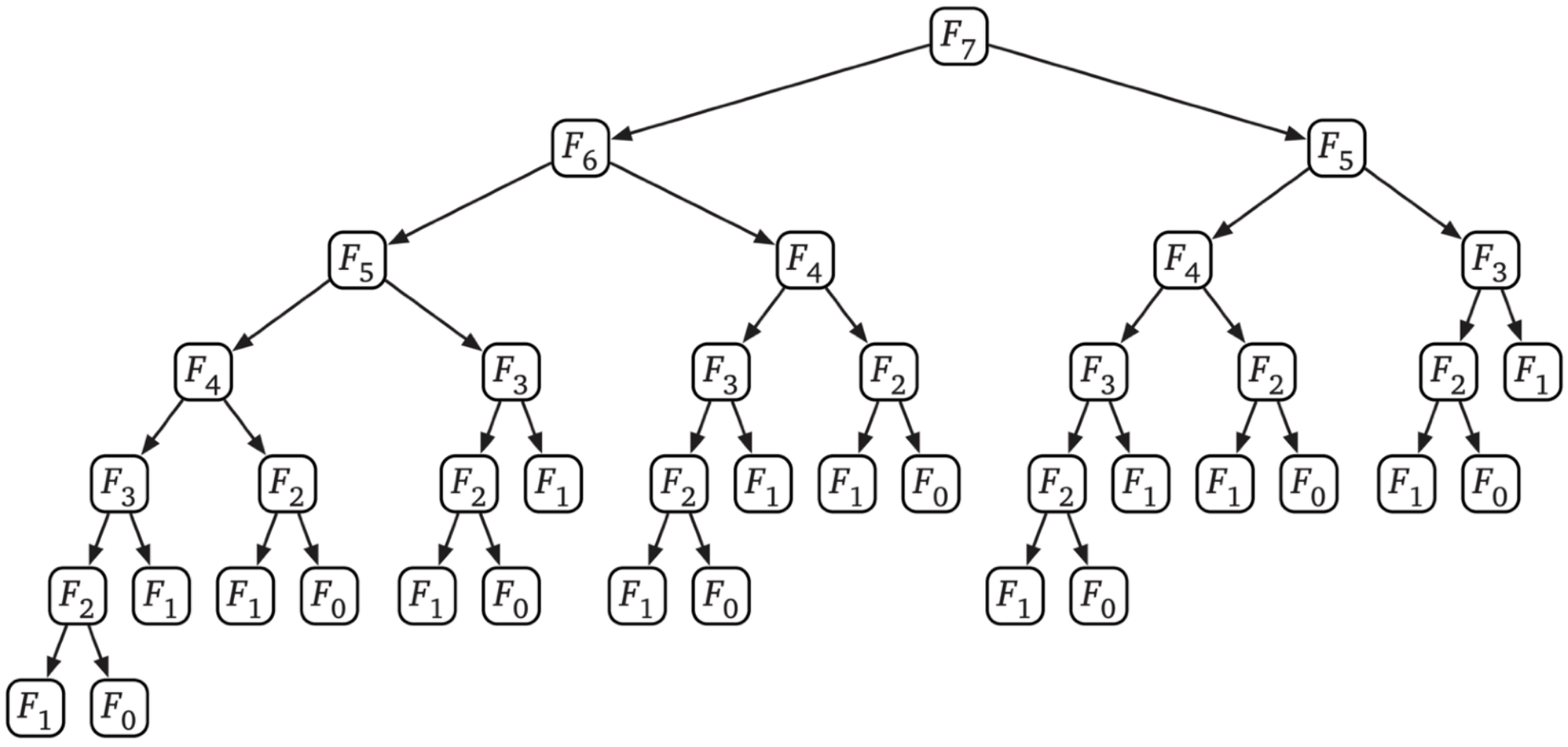
Slow Recursion: Fibonnacci

- Correct answer:
- $T(n) \geq F_n$ for all $n \geq 1$
- $F_n \geq \phi^{n-2}$ where $\phi = \left(\frac{1+\sqrt{5}}{2}\right) \approx 1.6^{n-2}$ (exponential!)

```
RecFibo(n):
    if n = 0
        return 0
    else if n = 1
        return 1
    else
        return RecFibo(n − 1) + RecFibo(n − 2)
```

Memo(r)ization

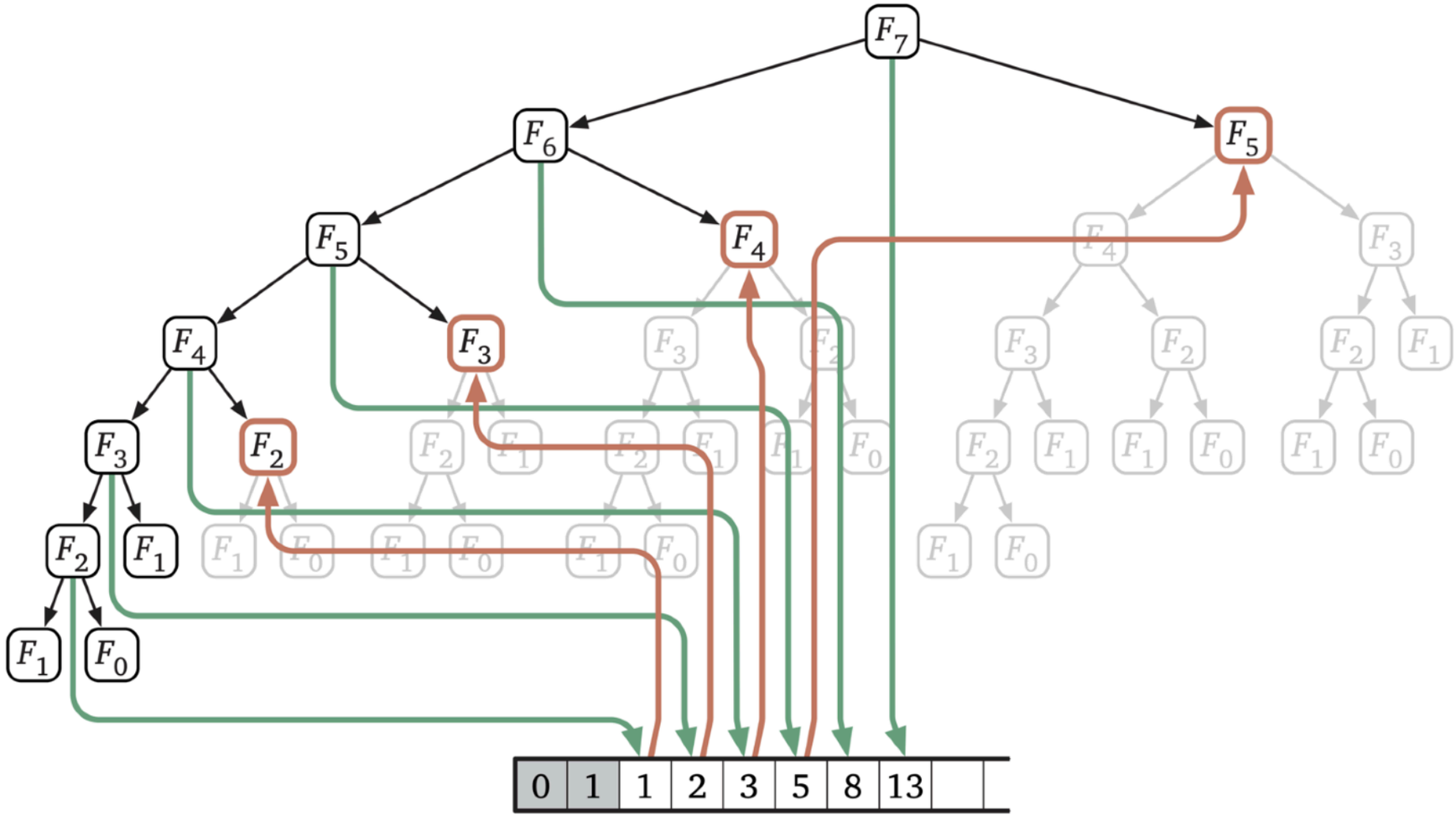
- Recursive Fibonacci algorithm is slow because it computes the same functions over and over
- Can speed it up considerably by writing down the results of our recursive calls, and looking them up when we need them later



Dynamic Programming: Smart Recursion

- Dynamic programming is all about smart recursion by using memoization
- Here it cuts down on all useless recursive calls

$$T[n] = T[n-1] + T[n-2] + 1$$



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

Memoizing Fibonacci

- Write each entry down in an array when you compute it
- How do we compute the nth Fibonacci number?
  - Fill in the first two Fibonacci numbers.
  - Use those to fill in the third, then fourth, etc.
  - Takes $O(1)$ to fill in a table entry
  - $O(n)$ overall

$$A = 1 \mid 1 \mid 2 \mid 3 \mid 5 \mid 8 \mid 13 \mid 21$$

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

Recipe for a Dynamic Program

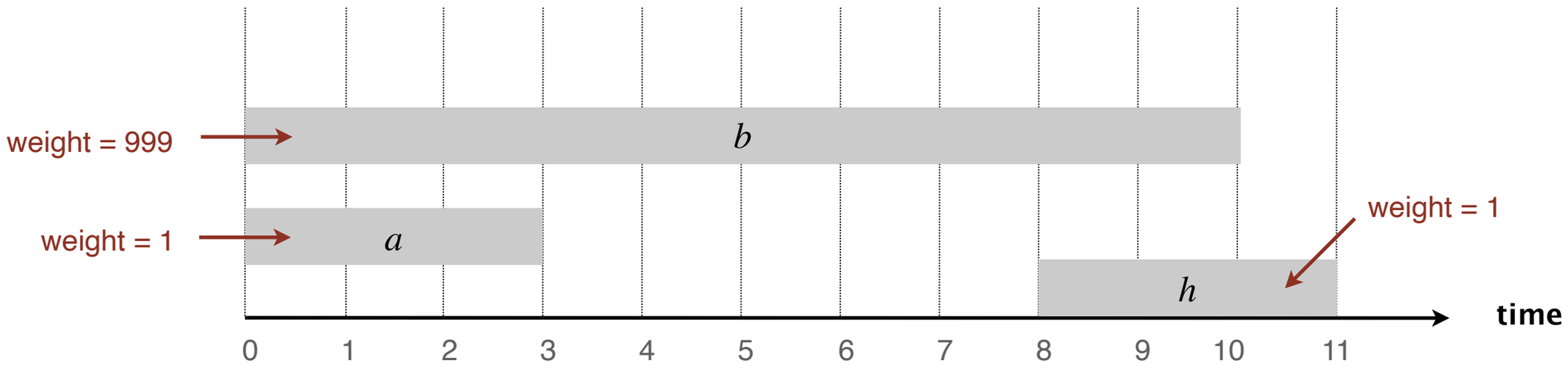
- **Formulate the right subproblem.** The subproblem must have an optimal substructure
- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem
- **State the base case(s)**. The subproblem thats so small we know the answer to it!
- **State 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

- **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
- **Output.** We must select non-overlapping intervals with the maximum weight. That is, output $I \subseteq \{1,\ldots,n\}$ that are pairwise non-overlapping that maximize $\sum_{i \in I} v_i$
- **Optimal cost.** Can we just find the value of the best solution? Find the largest $\sum_{i \in I} v_i$ where intervals in I are compatible.
- Let Opt-Schedule(n) be the **value** of the optimal schedule

Remember Greedy?

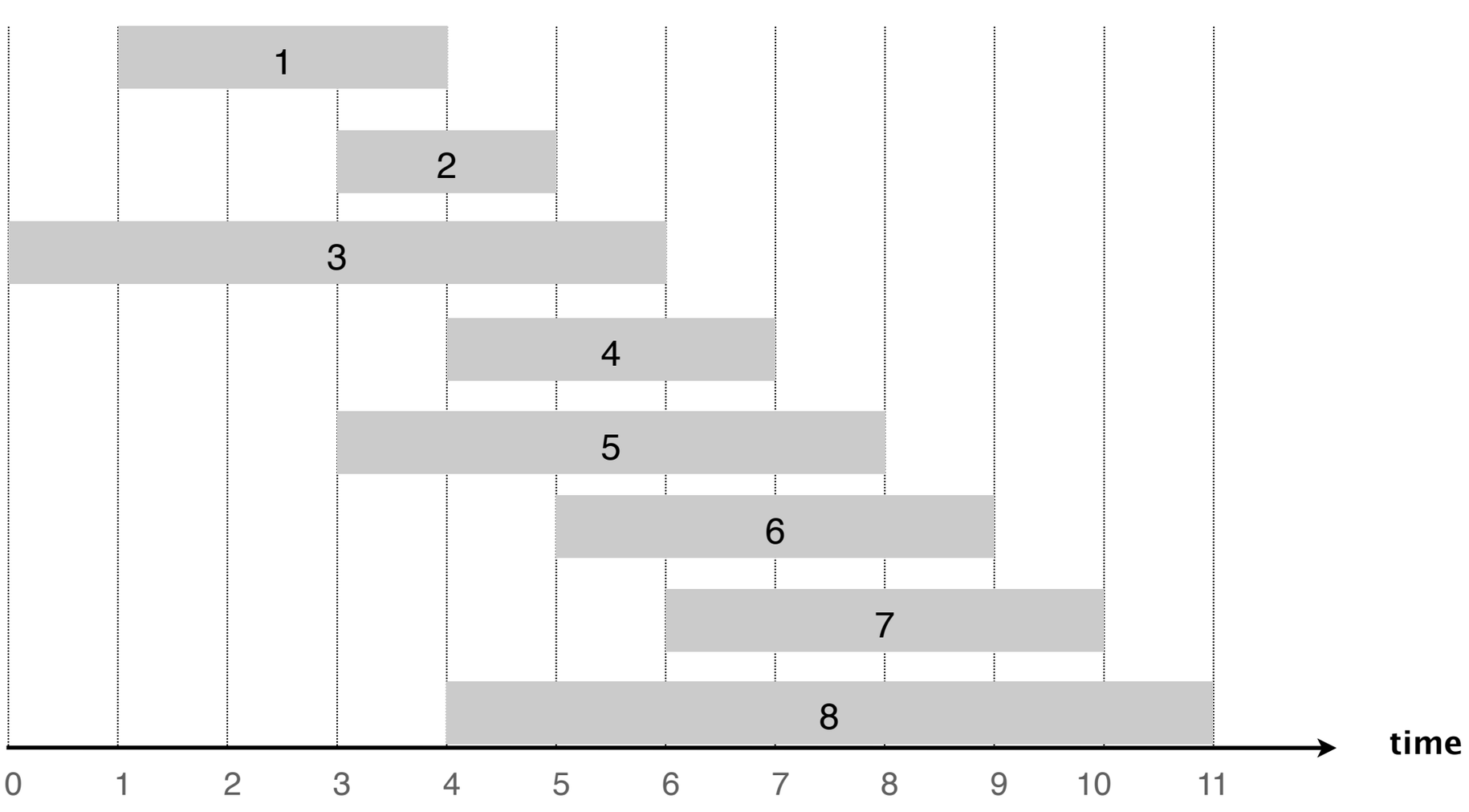
- Greedy algorithm earliest-finish-time first
    - Considers jobs in order of finish times
    - Greedily picks jobs that are non-overlapping
- We proved greedy is optimal when all weights are one
- How about the weighted interval scheduling problem?



Greedy fails spectacularly

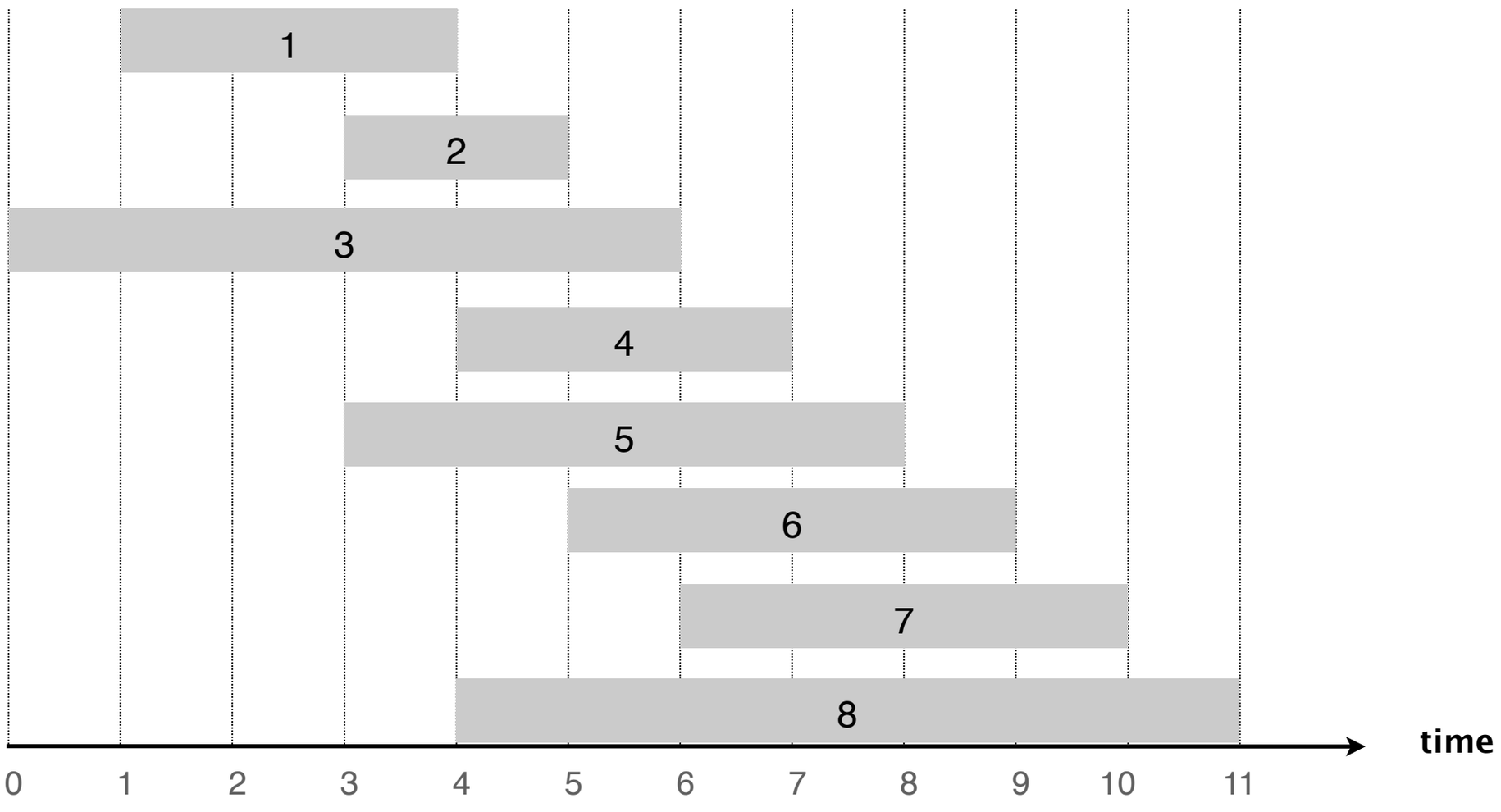
Helpful Information

- 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



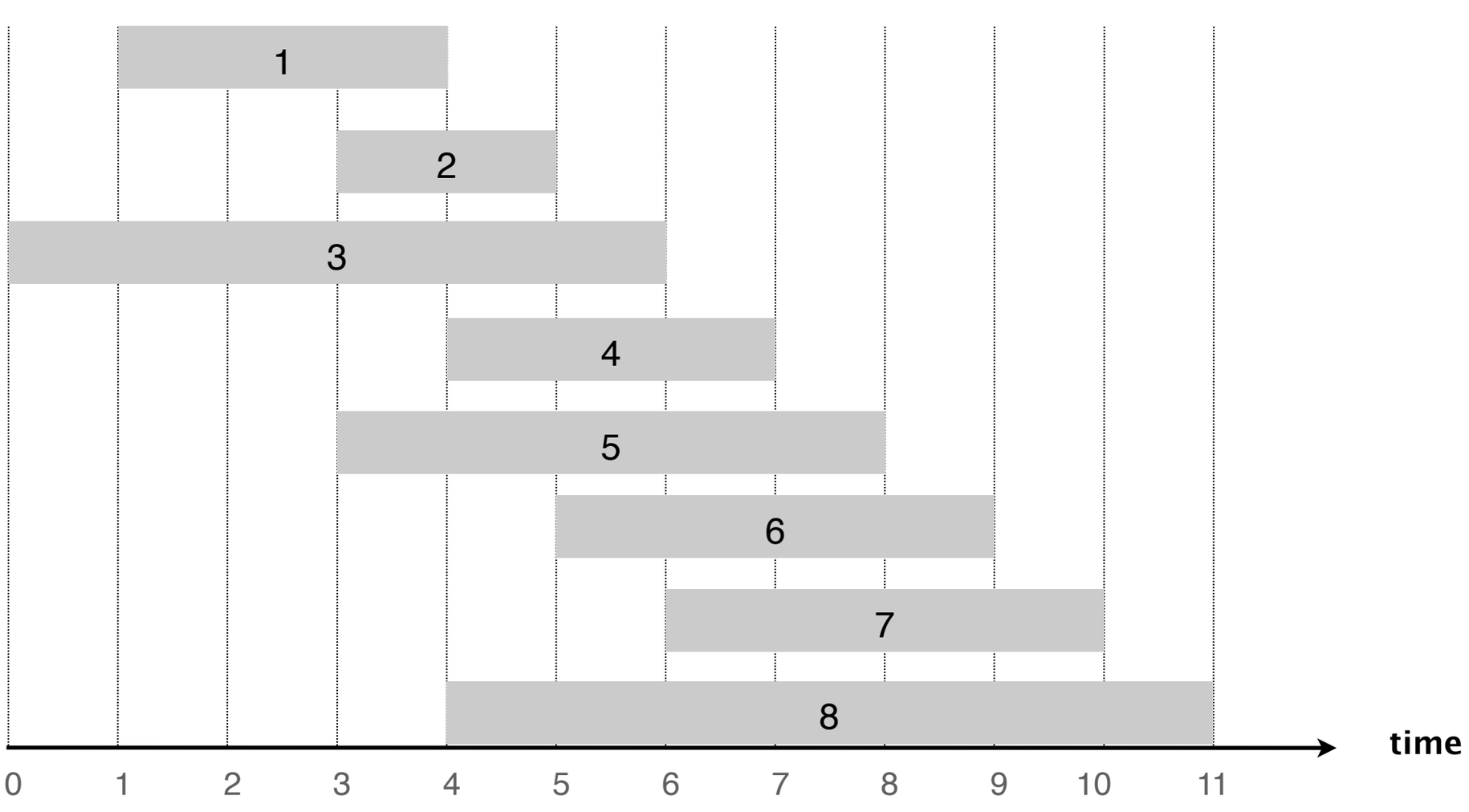
Helpful Information

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



Subproblem for our DP

- **Subproblem**.
  - For $1 \leq i \leq n,$ let Opt-Schedule(i) be the value of the optimal schedule that only uses intervals $\{1,\ldots,i\}$
  - Notice the optimal substructure
- Figuring out how we can build from smaller subproblems
  - Let us consider the last interval i with $(s_i,\ t_i)$
    - **Case 1.** Interval i is not in the optimal solution, then Opt-Schedule(i) = Opt-Schedule$(i-1)$
    - **Case 2.** Interval i is in the optimal solution
    - No two intervals in the schedule can overlap: cannot have $j < i$ such that $s_i \leq f_j$
    - Only intervals $j \leq p(i)$ can be in the same schedule as i

Recurrence for our DP

- **Subproblem**.
    - For $1 \leq i \leq n,$ 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

Finding $p(i)$

- Can do a linear scan in $O(i)$ time
- Or: we have the intervals in sorted order by finishing time. Binary search for s_i (the start time of i) in this list
  - Finds the largest interval j with $f_j \leq s_i$
  - Then $p(i) = j$
  - Time is $O(\log i) = O(\log n)$

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
  - $O(\log n)$ to find $p(i)$
- Do we need to do any preprocessing?
  - Need to sort; $O(n \log n)$
- Overall running time: $O(n \log n)$

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!

Recursive Solution?

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

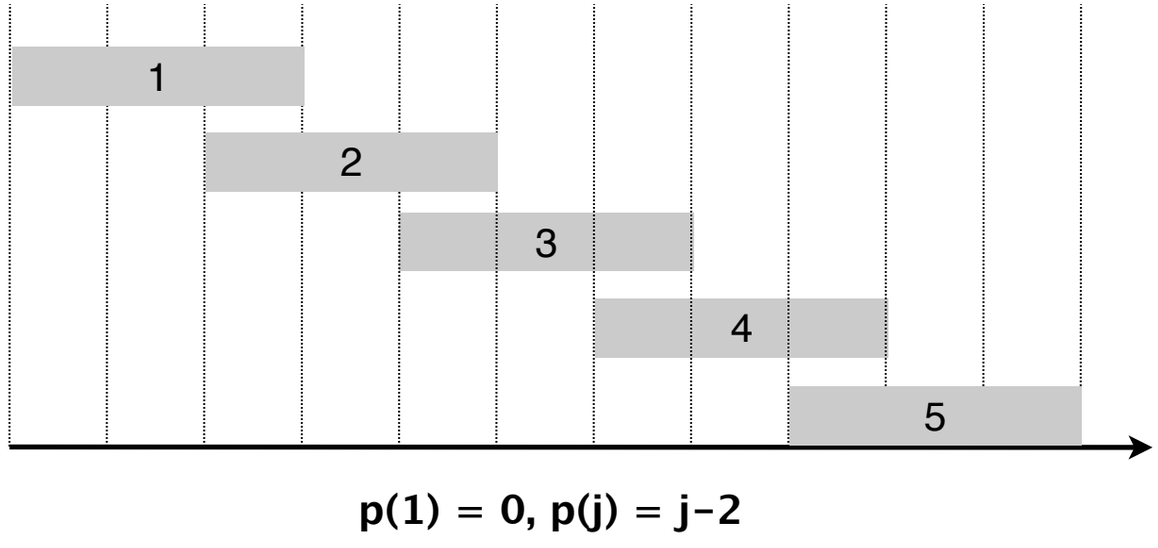
Opt-Schedule(i):

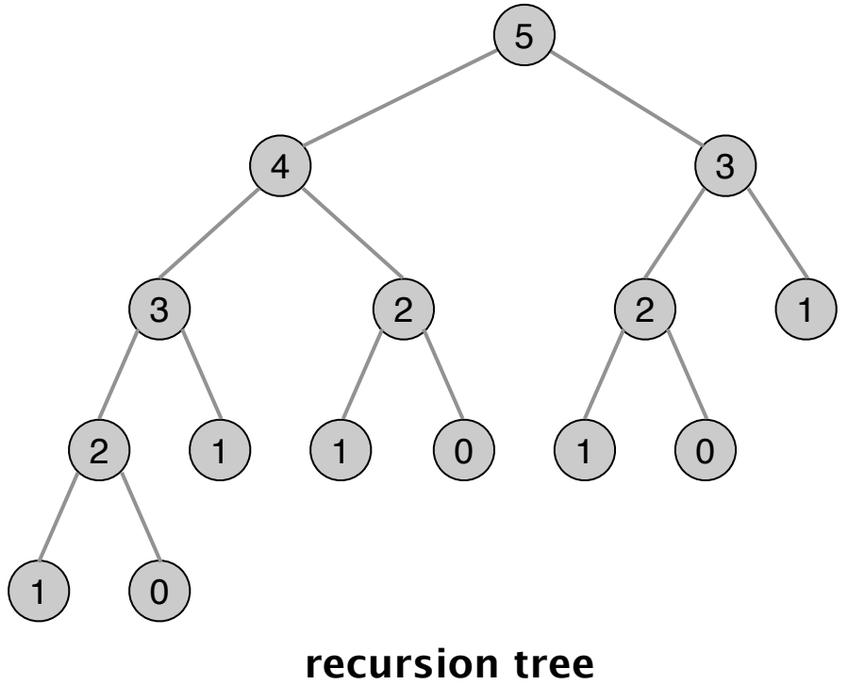
- If $j = 0$, return 0
- Else
  - Return $\mathbf{max}($Opt-Schedule$(j-1), v_j +$ Opt-Schedule$(p(j)))$

- 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 time!
- Lots of redundancy!
    - How many distinct subproblems are there to solve?
    - Opt-Schedule(i) for $1 \leq i \leq n+1$





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

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)