Week 8 - Friday

COMP 4500

Last time

- What did we talk about last time?
- Exam 2!
- And before that?
  - Review
- And before that?
  - Master Theorem
  - Solved exercises from Chapter 5

Questions?

Exam 2 Post Mortem

Dynamic Programming

Previous approaches

- We covered **greedy** approaches, where you simply want to take the next best thing
  - These are often linear or $O(n \log n)$ due to sorting
- We looked at **divide-and-conquer** approaches
  - Usually taking an unimpressive polynomial running time like $O(n^2)$ and improving it, perhaps to $O(n \log n)$
- But there are harder problems that appear as if they might take exponential time

Dynamic Programming

- **Dynamic programming** shares some similarities with divide and conquer
  - We break a problem down into **subproblems**
  - We build correct solutions up from smaller subproblems into larger ones
- The subproblems tend not to be as simple as simply dividing the problem in half
- Dynamic programming dances on the edge of exploring an exponential number of solutions
  - But somehow manages to look at only a polynomial set!

Weighted Interval Scheduling

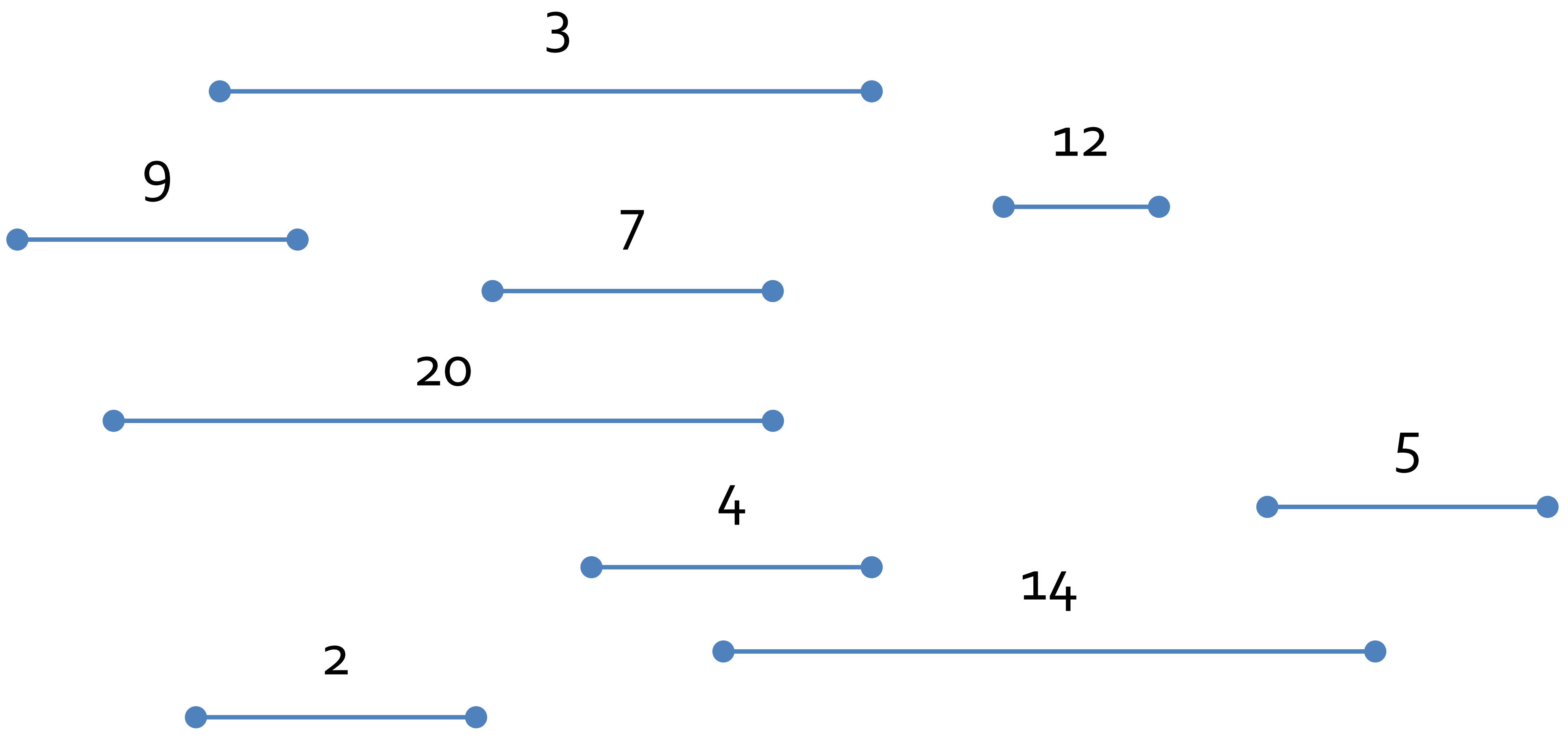
Interval scheduling

- In the interval scheduling problem, some resource (a phone, a motorcycle, a toilet) can only be used by one person at a time.
- People make requests to use the resource for a specific time interval [s, f].
- The goal is to schedule as many uses as possible.
- There's no preference based on who or when the resource is used.

Weighted interval scheduling

- The **weighted interval scheduling** problem extends interval scheduling by attaching a weight (usually a real number) to each request
- Now the goal is not to maximize the **number** of requests served but the total **weight**
- Our greedy approach is worthless, since some high value requests might be tossed out
- We could try all possible subsets of requests, but there are **exponential** of those
- **Dynamic programming** will allow us to save parts of optimal answers and combine them efficiently

Weighted interval scheduling example



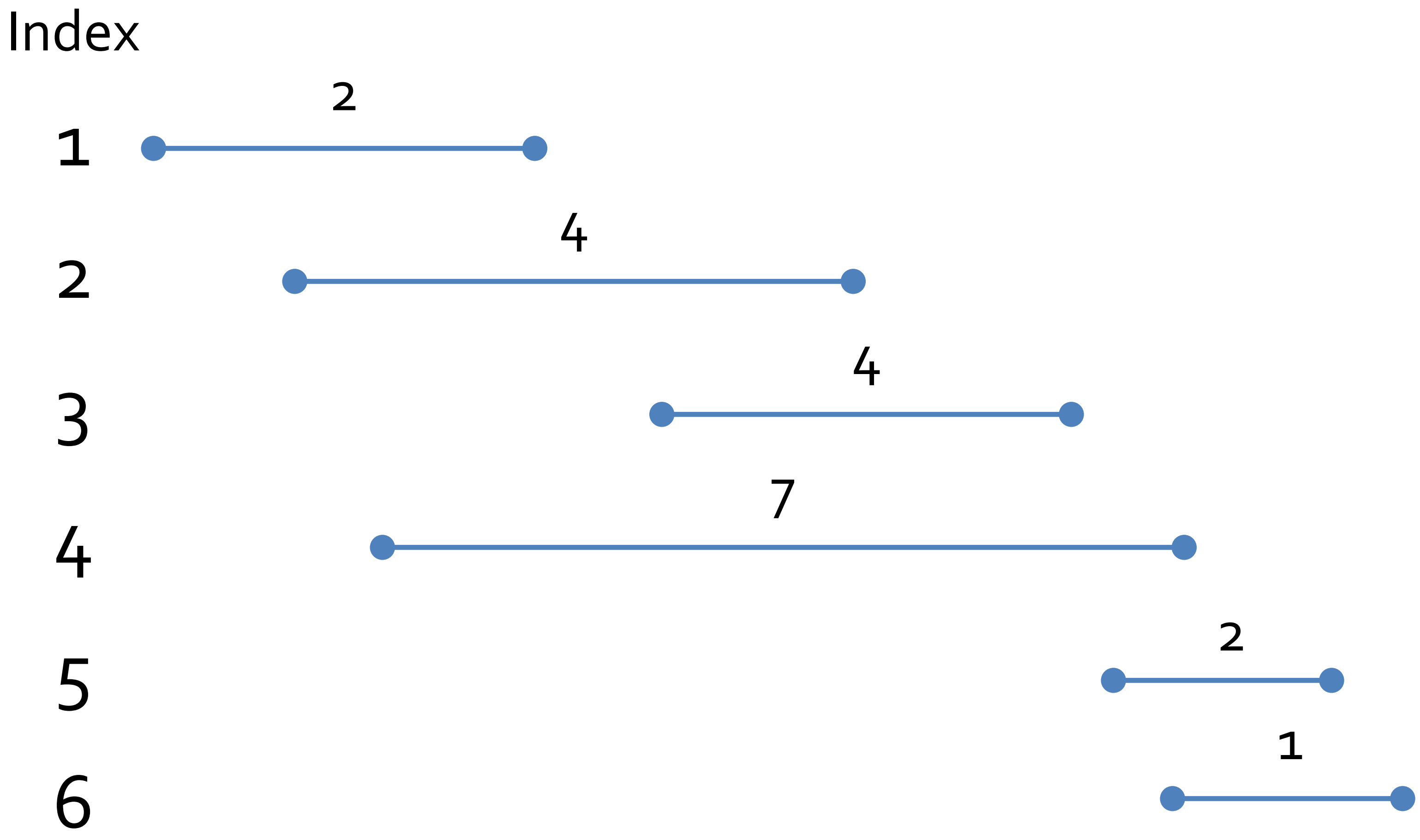
Notation

- We have n requests labeled 1, 2,..., n
- Request i has a start time s_i and a finish time f_i
- Request i has a value v_i
- Two intervals are **compatible** if they don't overlap

Designing the algorithm

- Let's go back to our intuition from the unweighted problem
- Imagine that the requests are sorted by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$
- We say that request i comes before request j if $i < j$, giving a natural left-to-right order
- For any request j, let $p(j)$ be the largest index $i < j$ such that request i ends before j begins
  - If there is no such request, then $p(j) = 0$

$p(j)$ examples



$p(1) = 0$

$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$

More algorithm design

- Consider an optimal solution ***O***
  - It either contains the last request ***n*** or it doesn't
- If ***O*** contains ***n***, it does not contain any requests between ***p***(***n***) and ***n***
- Furthermore, if ***O*** contains ***n***, it has an optimal solution for the problem for just requests 1, 2, …, ***p***(***n***)
  - Since those requests don't overlap with ***n***, they have to be the best or they wouldn't be optimal
- If ***O*** does not contain ***n***, then ***O*** is simply the optimal solution of requests 1, 2,…, ***n*** - 1

Subproblems found!

- It might not be obvious, but the last slide laid out a way to break a problem into smaller subproblems
- Let OPT(j) be the value of the optimal solution to the subproblem of requests 1, 2,…, j
- $\text{OPT}(j) = \max(v_j + \text{OPT}(p(j)), \text{OPT}(j - 1))$
- Another way to look at this is that we will include j in our optimal solution for requests 1, 2,…,j

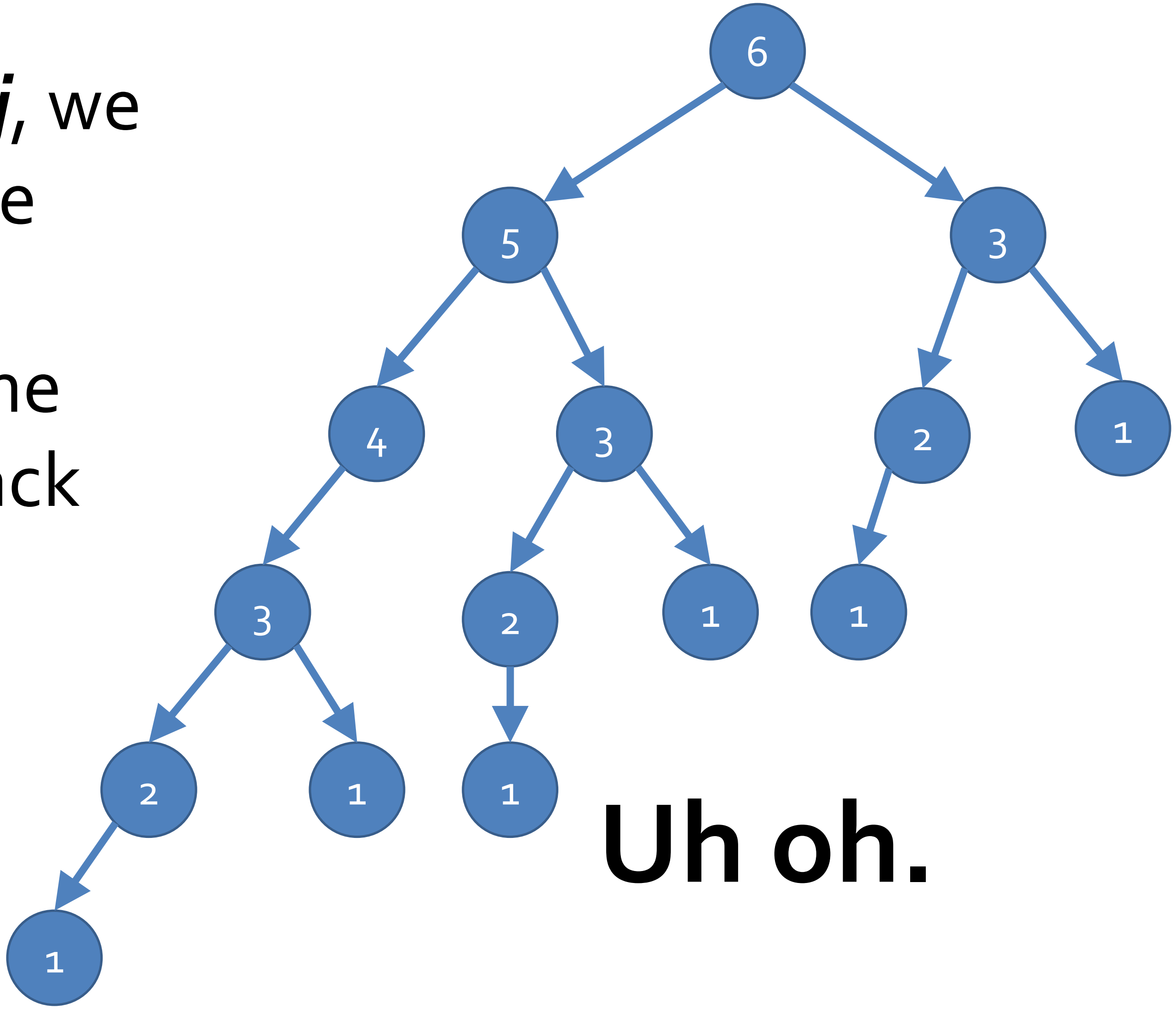
iff $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$

We've already got an algorithm!

- Compute-Opt(j)
  - If $j = 0$ then
    - Return 0
  - Else
    - Return max(v_j + Compute-Opt($p(j)$), Compute-Opt($j - 1$))

How long does Compute-Opt take?

- Well, for every request *j*, we have to do two recursive calls
- Look at the tree from the requests a few slides back



Uh oh.

Needless recomputation

- The issue here is that we are needlessly recomputing optimal values for smaller subproblems
- You might recall that we had a similar problem in COMP 2100 with the naïve implementation of a recursive Fibonacci function
- In the worst case, the algorithm has an exponential running time
- Just **how** exponential depends on the structure of the problem

Upcoming

Next time…

- Finish weighted interval scheduling
- Segmented least squares
- **No class next week!**

Reminders

- For after spring break:
  - Read sections 6.2 and 6.3