Midterm Review

Asymptotic runtime analysis

- $O(\le)$, $\Theta(=)$, $\Omega(\ge)$
  - Simplifying complex functions to ease the asymptotic comparisons
  - Familiar with basic facts. e.g., $\log_a x = \log_a b \log_b x$, $a^{\log_b x} = x^{\log_b a}$
  - See my email to class list summarizing the main strategies for deciding asymptotic orders

- Characterize runtime based on pseudo-code

- Forming recurrence equation for recursive algo.

- Solving recurrence equations
  - Telescoping
  - Recursion tree
  - Master theorem

Divide and Conquer

- Break problem into smaller sub-problems
- Solve smaller sub-problems via recursion
- Combine solutions of sub-problems to get a solution to the original problem
- Examples:
    - Merge sort (n log n)
    - Binary search (log n)

Majority element problem

- Given an array of n elements $a_1, a_2, \ldots, a_n$
- An element is majority if it occurs more than $\frac{n}{2}$ times
- You cannot sort the array, but can compare two elements to see if they are the same in O(1) time
- Goal: find the majority element if it exists in O(n log n) time.

High level idea

- Break A into A_L and A_R
- We can recursively find the majority element in A_L and A_R – call them m_L and m_R
- For possible outcomes:
  1. $m_L = m_R = \text{NULL}$
  2. $m_L = m_R \neq \text{NULL}$
  3. $m_L \neq \text{NULL}, m_R = \text{NULL}$
  4. $m_L = \text{NULL}, m_R \neq \text{NULL}$
- Key insight: if an element is majority of A, it has to be majority for either A_L and A_R
  - Otherwise, its total occurrence would be $\leq n/2$

```
Majority(A, n)

A_L= A(1,…,n/2)
A_R= A(n/2,…,n)
m_L=majority(A_L,n/2)
m_R=majority(A_R,n/2)
if m_L = m_R
      return m_L
if m_L ≠ NULL
     scan A to see if m_L occurs >n/2
      return m_L if yes
if m_R ≠ NULL
      scan A to see if m_R occurs >n/2
      return m_R if yes
return NULL
```

What is missing?

Correctness proof (induction)

Base case:

Inductive assumption:

Inductive step:

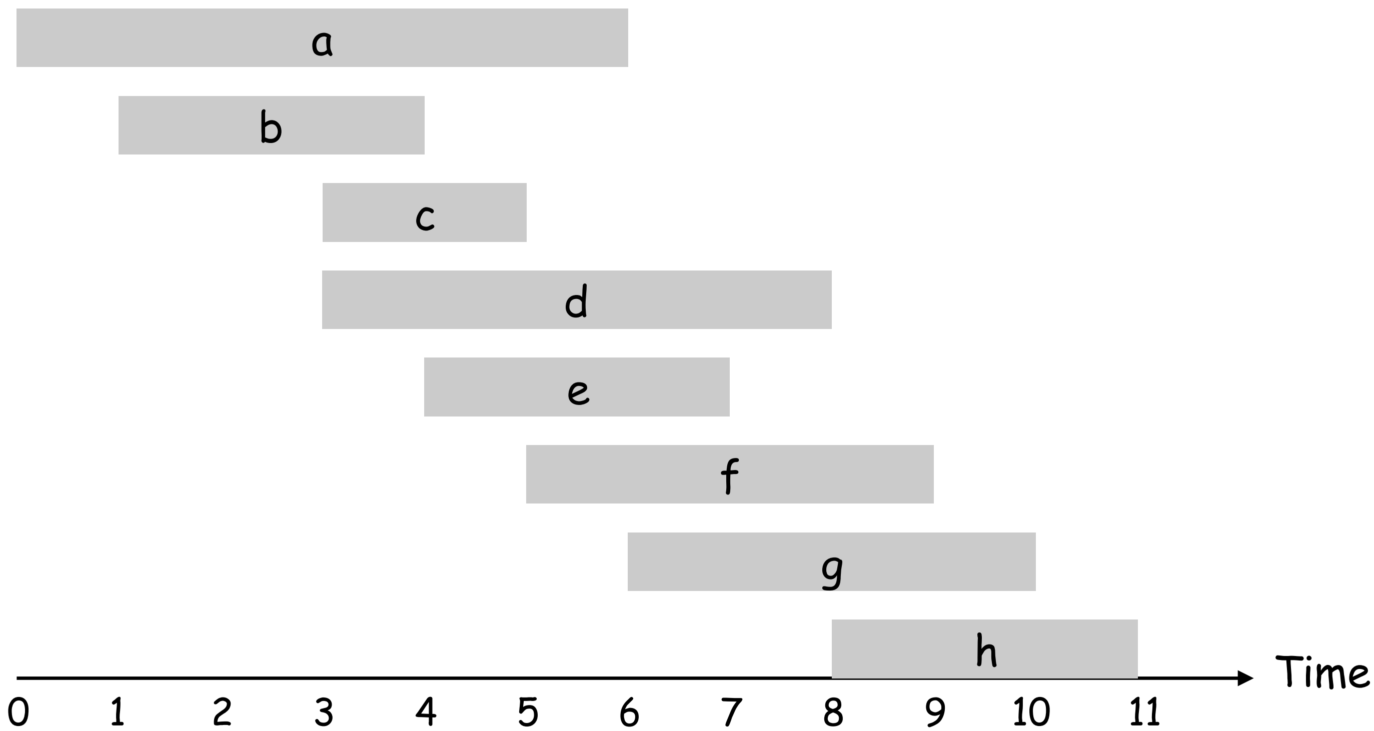
Dynamic programming

- Define subproblem
- Figure out the recursive relation for subproblem
- Work out the base cases and an iterative procedure to incrementally solve all subproblems
- Return the solution to the original problem

Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

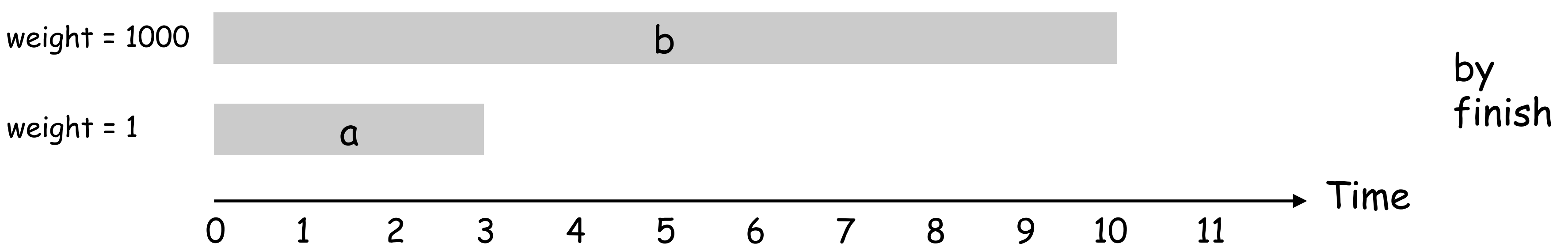


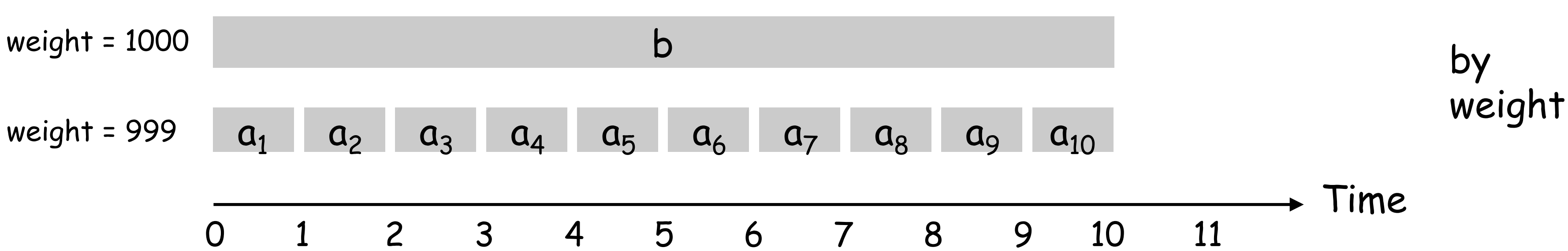
Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



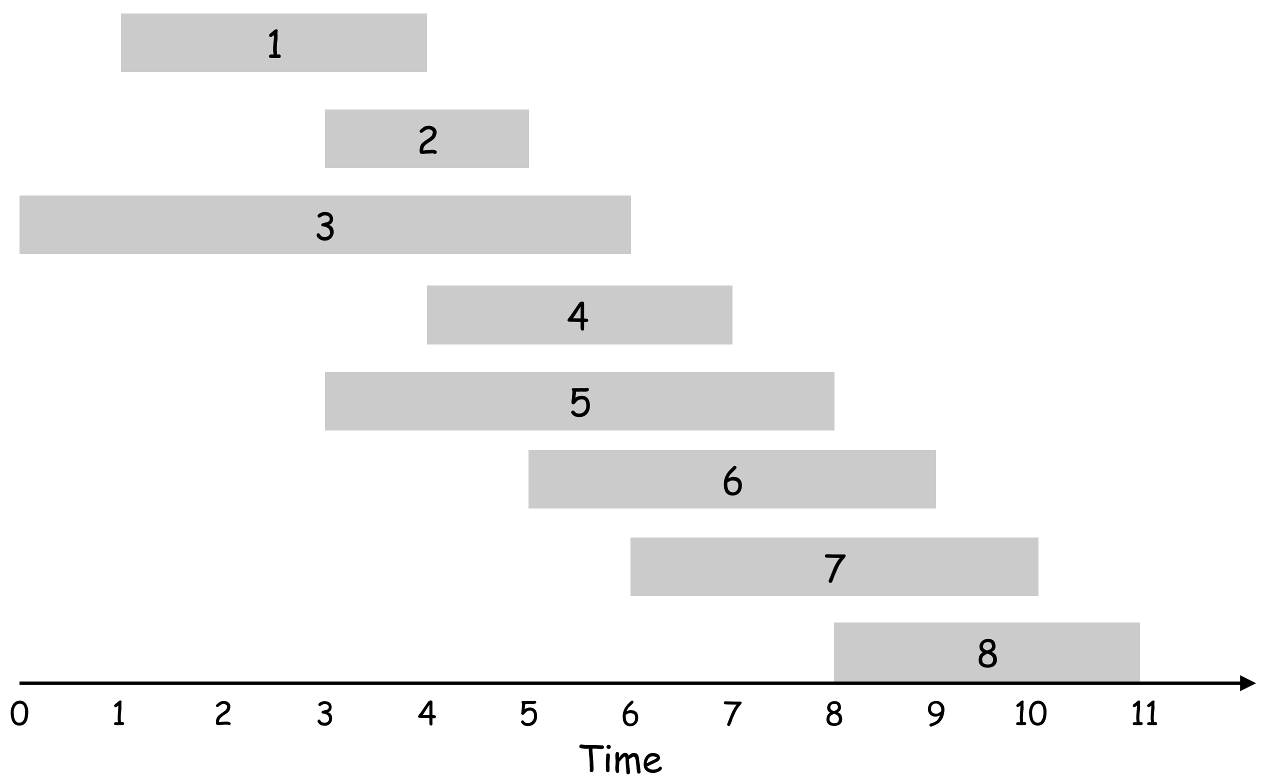


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



j	p(j)
0	-
1	0
2	0
3	0
4	1
5	0
6	2
7	3
8	5

Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Option 1: selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Option 2: does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

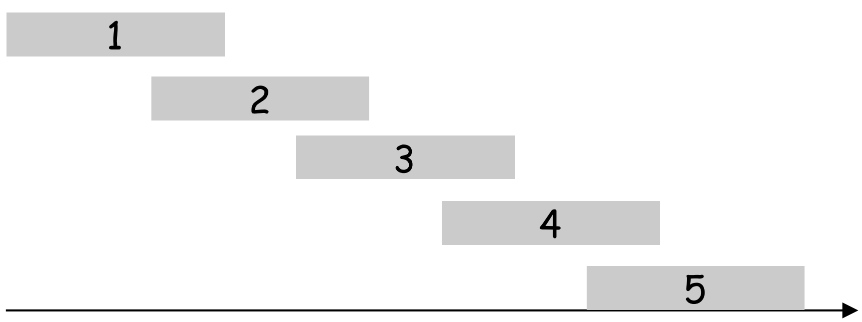
Brute force recursive algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

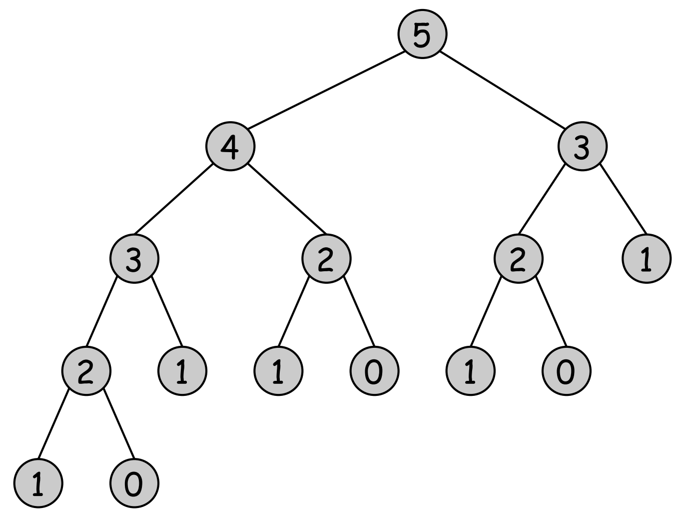
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



$p(1) = 0, p(j) = j-2$



Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   OPT[0] = 0
   for j = 1 to n
      OPT[j] = max(v_j + OPT[p(j)], OPT[j-1])
}

Output OPT[n]
```

Claim: OPT[j] is value of optimal solution for jobs 1..j

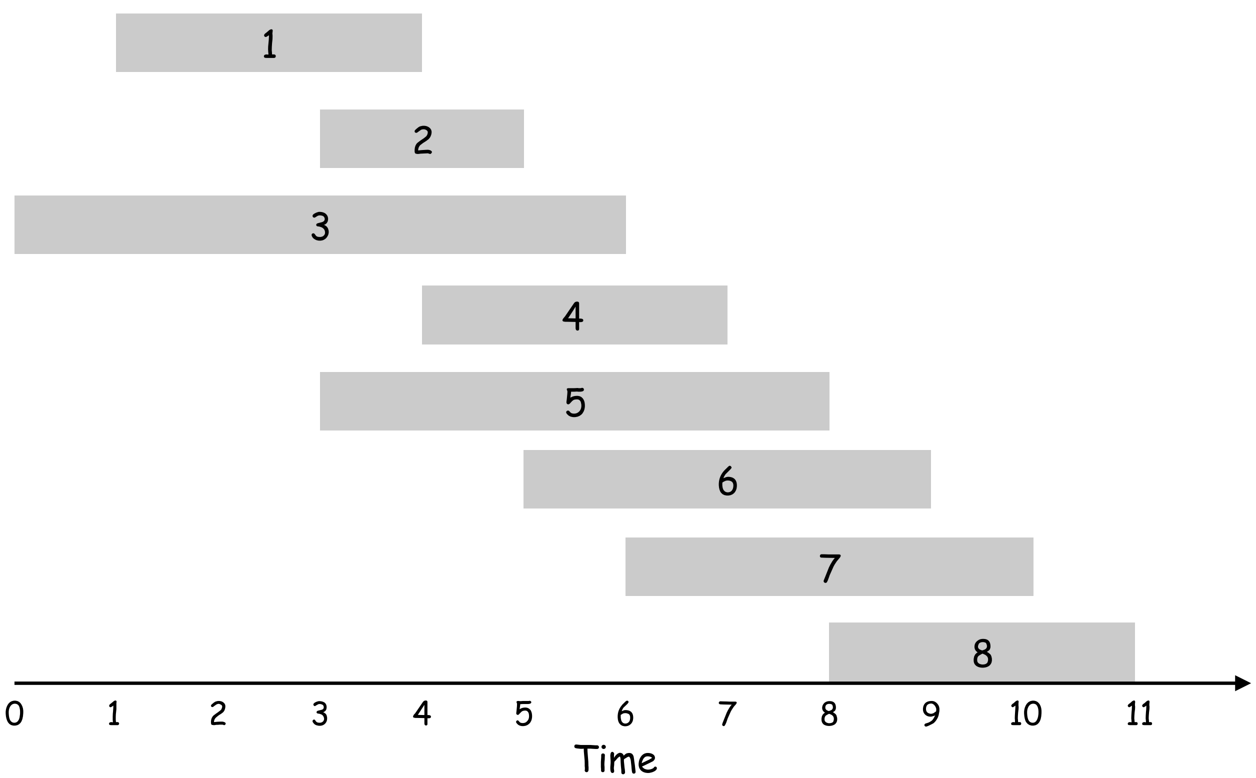
Timing: Easy. Main loop is $O(n)$; sorting is $O(n \log n)$

Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



j	vj	pj	optj
0	-	-	0
1		0	
2		0	
3		0	
4		1	
5		0	
6		2	
7		3	
8		5	