CS-3510:
Design and Analysis of Algorithms

Greedy Algorithms

Instructor: Shahrokh Shahi

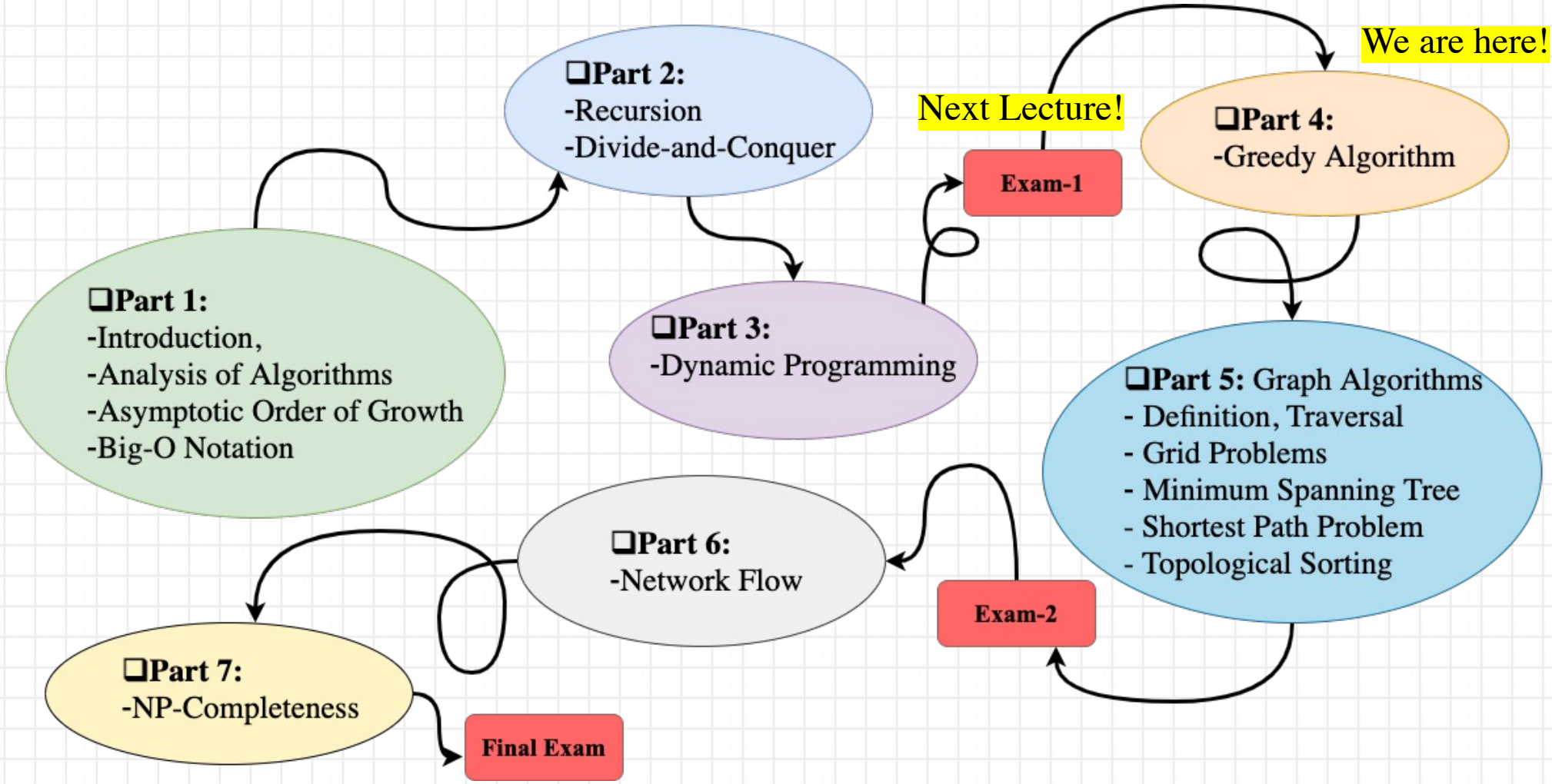
College of Computing
Georgia Institute of Technology
Summer 2022

Announcements

- HW 3 is released; the submission deadline is extended for one week
  - Due next Friday, June 17

- Exam on Thursday at 3:30
  - Closed-book, one sheet of notes
  - 80 minutes
  - 5 main questions
    - Q1: Asymptotic Notations
    - Q2: Master Theorem
    - Q3: Divide-and-Conquer
    - Q4: Divide-and-Conquer
    - Q5: Dynamic Programming

Roadmap


❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 2:
-Recursion
-Divide-and-Conquer
❑Part 3:
-Dynamic Programming
Exam-1
Next Lecture!
We are here!
❑Part 4:
-Greedy Algorithm
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

Greedy Algorithms

- Build the solution step-by-step
- At each step, make a decision that is locally optimal
- Never look back and hope for the best!
- Do NOT always yield optimal solutions, but for many problems they do

Greedy Choice Property

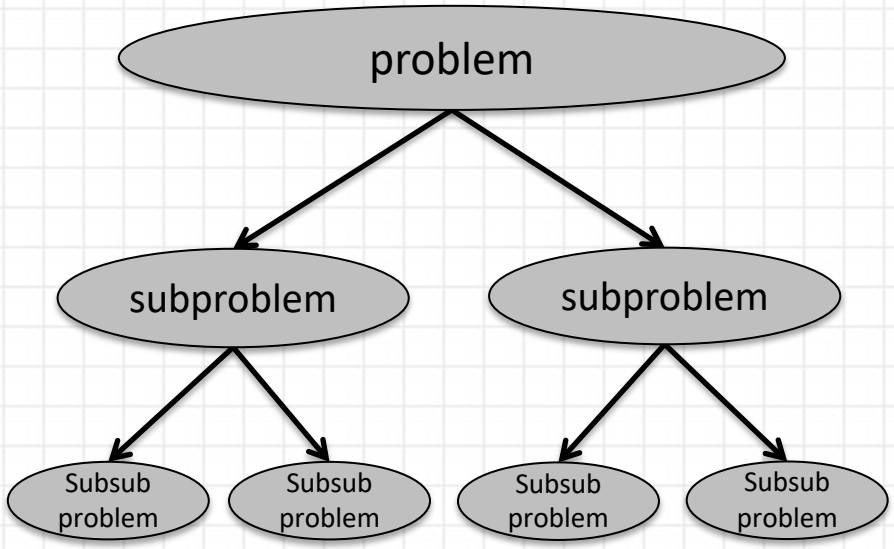
- Greedy choice = locally optimal choice
- Greedy-choice property: we can assemble a globally optimal solution by making locally optimal choices.
- In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.
  (The main difference with dynamic programming)
- Make whatever choice seems best at the moment and then solve the subproblem that remains.
- Makes its first choice before solving any subproblems.

Difference with Dynamic Programming

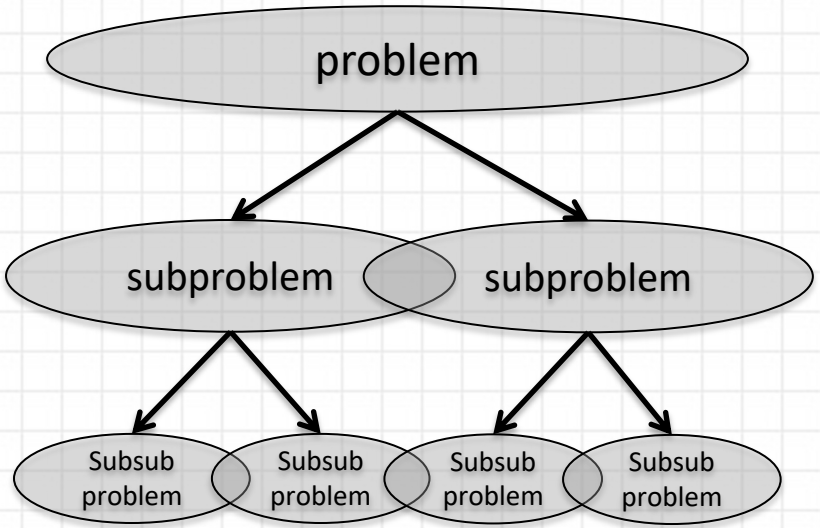
- Dynamic programming:
  - Make a choice at each step, but the choice usually depends on the solutions to subproblems.
  - Consequently, we typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems.
  - Even in top-down approach, we use memoizing. So, even though the code works top down, we still solve the subproblems before making a choice.
  - Solves the subproblems before making the first choice.

Greedy Algorithms

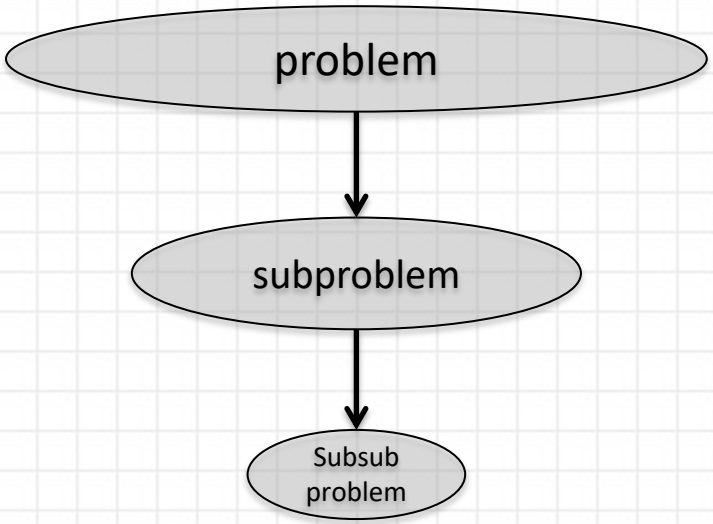
Divide-and-Conquer



Dynamic Programming

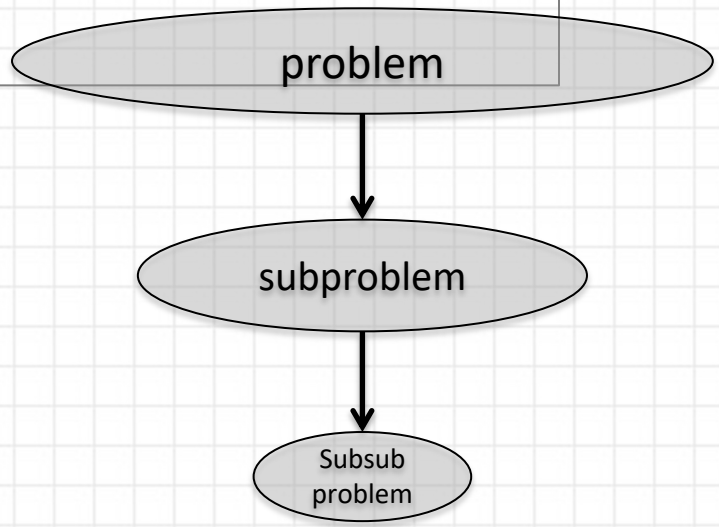


Greedy Approach



Optimal substructure
But only one subproblem

Greedy Algorithms



- Seems "easier" than dynamic programming?
- Two major "questions/problems":
  - What is the best/correct greedy choice to make?
  - How can we prove that the greedy algorithm yields an optimal solution?
- When is using the greedy approach a good idea?
  - Greedy can be optimal when the problem shows an especially nice optimal substructure.

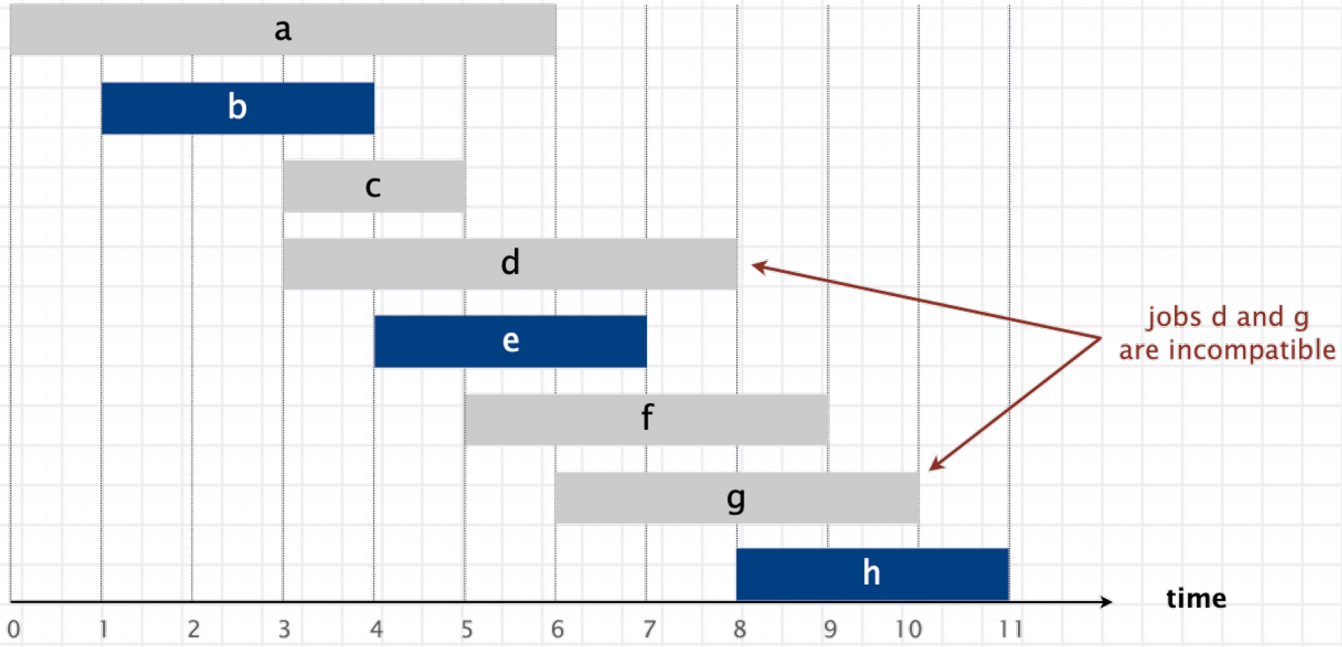
Greedy Algorithms

- Examples
  - Interval scheduling (activity selection)
  - Interval partitioning
  - Schedule to minimize lateness
  - …

- Applications in Graph (next week)
  - Kruskal's algorithm (minimum spanning tree)
  - Prim's algorithm (minimum spanning tree)
  - Dijkstra's algorithm (shortest path)

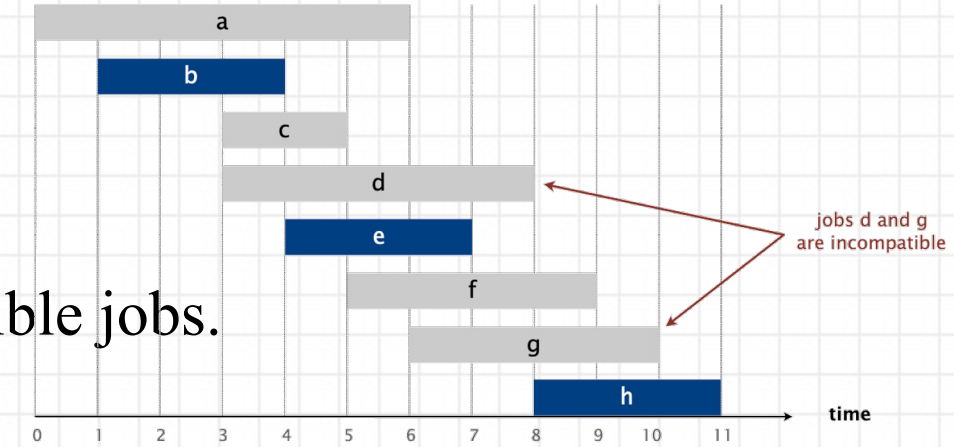
Greedy Example: (1) Interval Scheduling

- CLRS (16.1): Activity-selection problem
- KT (4.1): Interval-scheduling problem
- Problem
  - Job j starts at s_j and finishes at f_j.
  - Two jobs are compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.



Greedy Example: (1) Interval Scheduling

- Problem
  - Job j starts at s_j and finishes at f_j.
  - Two jobs are compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.

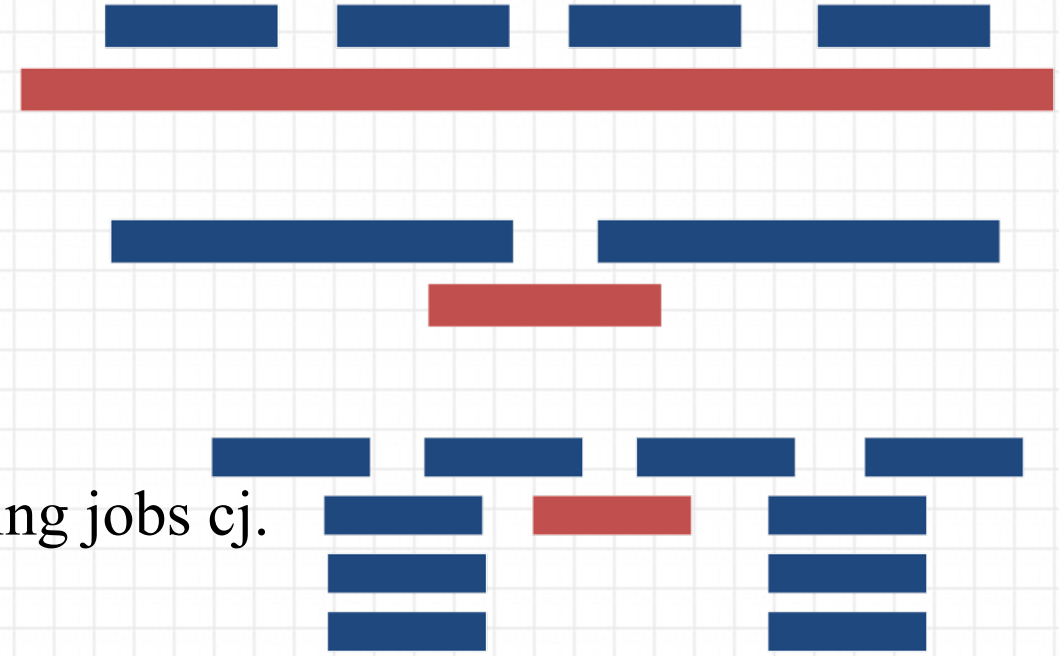


- Greedy approach:
  - Consider jobs in some natural order.
  - Take each job provided it's compatible with the ones already taken
    - [Earliest start time] Consider jobs in ascending order of s_j.
    - [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
    - [Fewest conflicts] For each job j, count the number of conflicting jobs cj. Schedule in ascending order of c_j.
    - [Earliest finish time] Consider jobs in ascending order of f_j.

Greedy Example: (1) Interval Scheduling

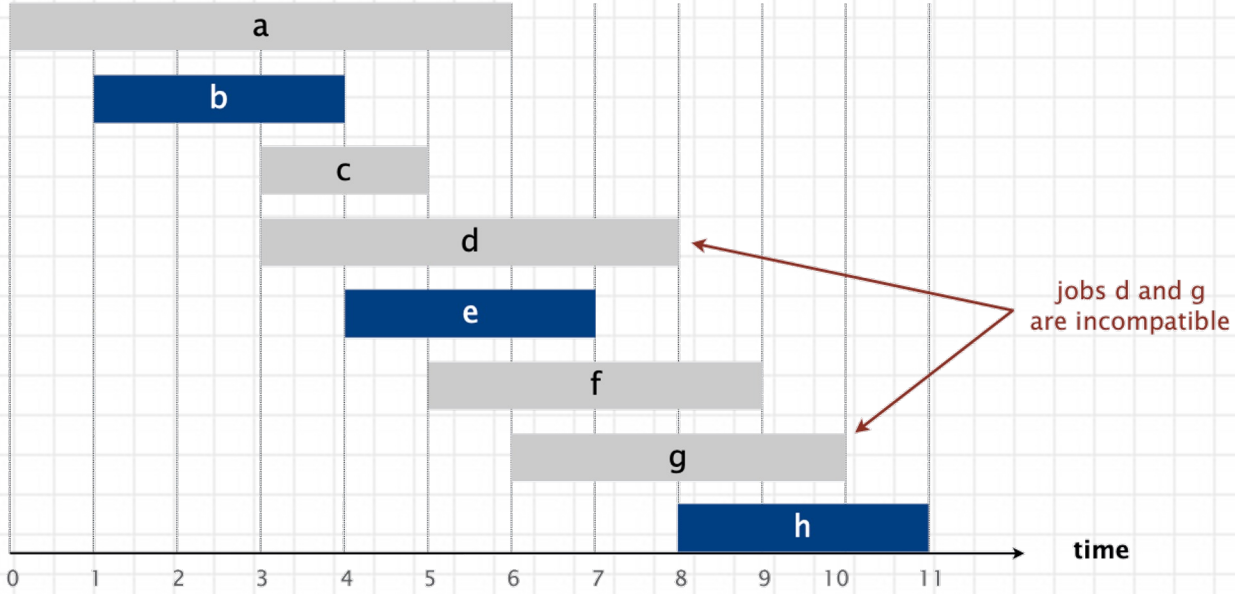
- Greedy approach:
  - × [Earliest start time]
    Consider jobs in ascending order of s_j.
  - × [Shortest interval]
    Consider jobs in ascending order of $f_j - s_j$.
  - × [Fewest conflicts]
    For each job j, count the number of conflicting jobs cj.
    Schedule in ascending order of c_j.
  - ✓[Earliest finish time] Consider jobs in ascending order of f_j.

Counterexamples



Greedy Example: (1) Interval Scheduling

- Problem
  - Job j starts at s_j and finishes at f_j.
  - Two jobs are compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.



- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

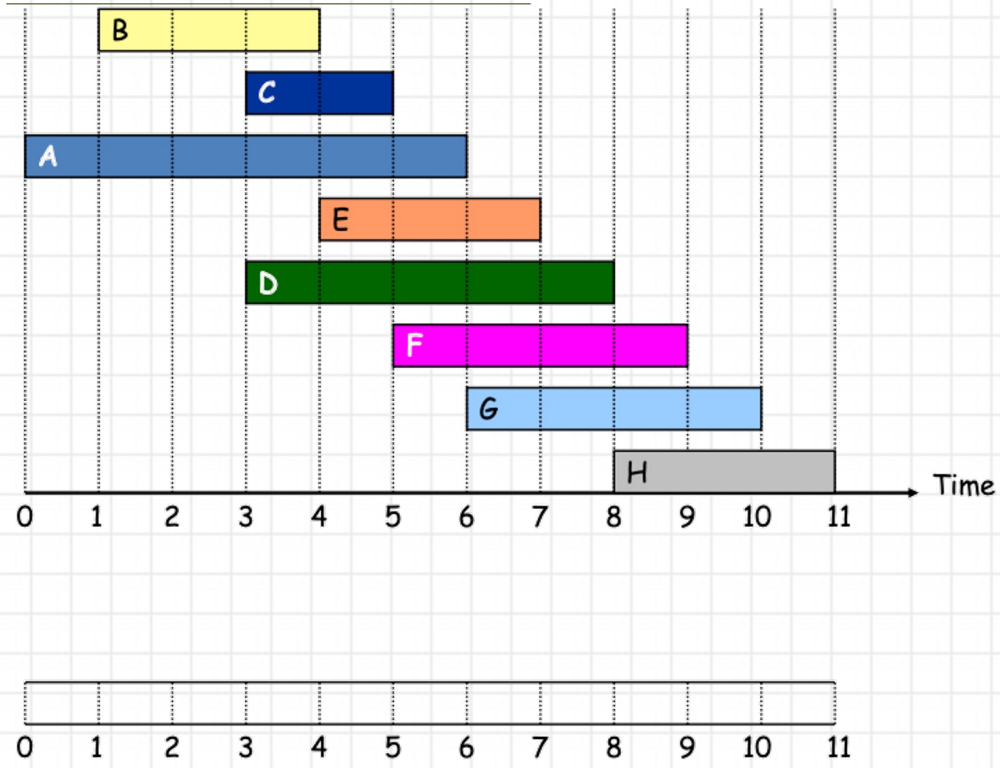
$S \leftarrow \varnothing$. ⟵ set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

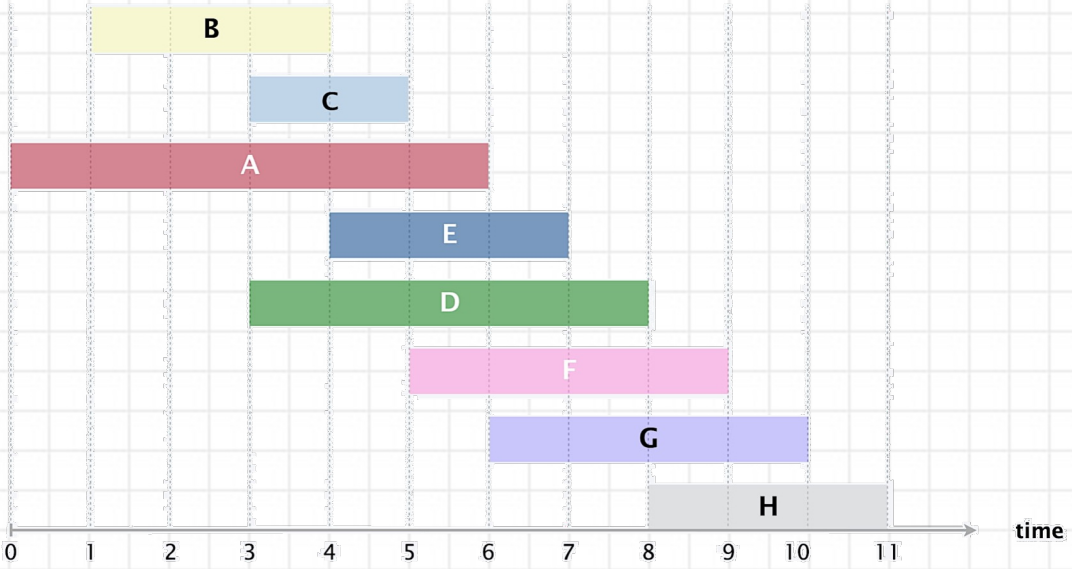
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

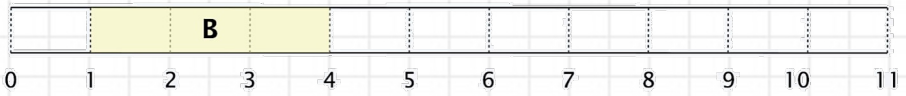
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job B is compatible (add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

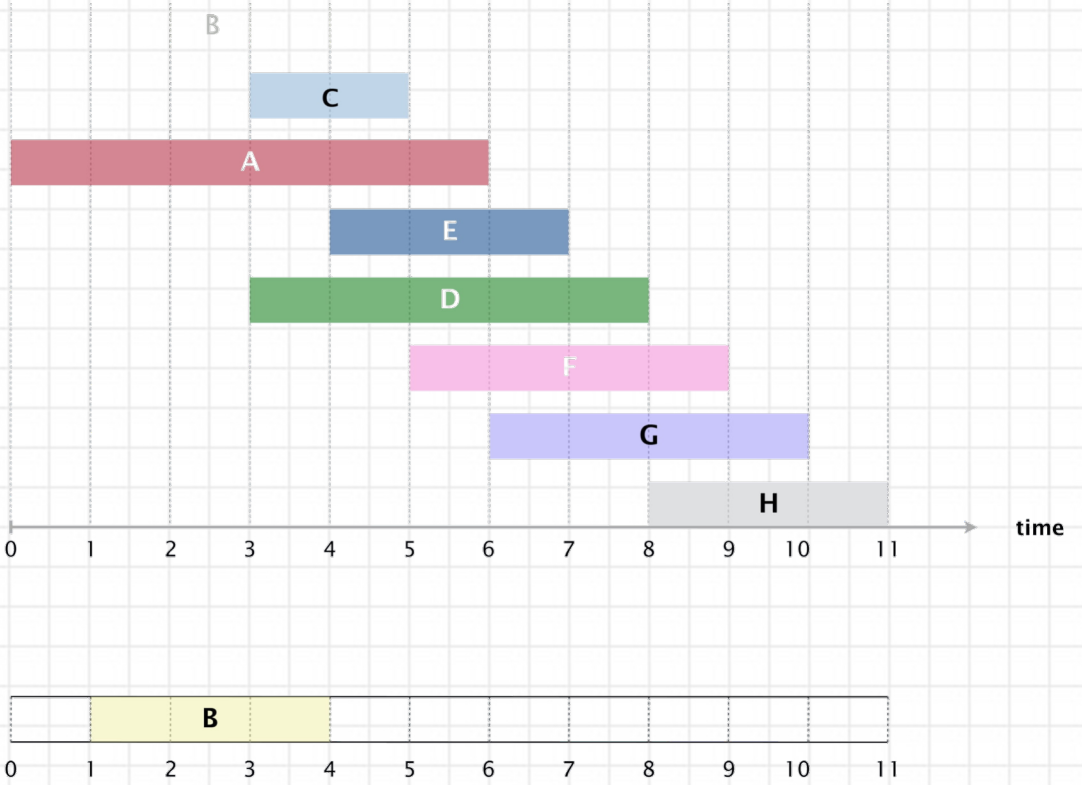
$S \leftarrow \varnothing$. ⟵ set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

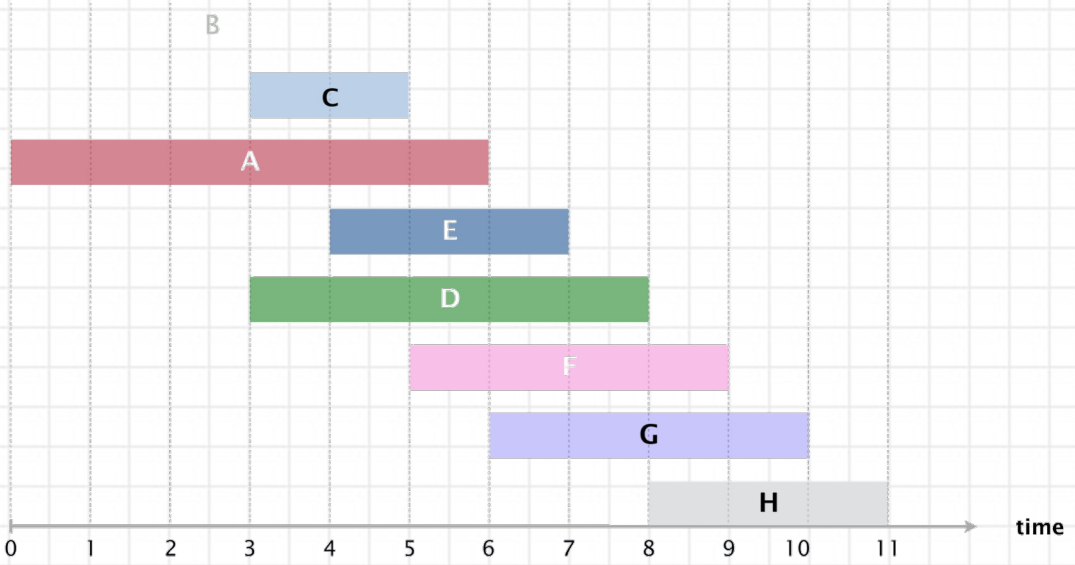
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

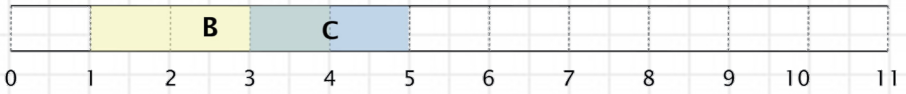
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job C is incompatible (do not add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

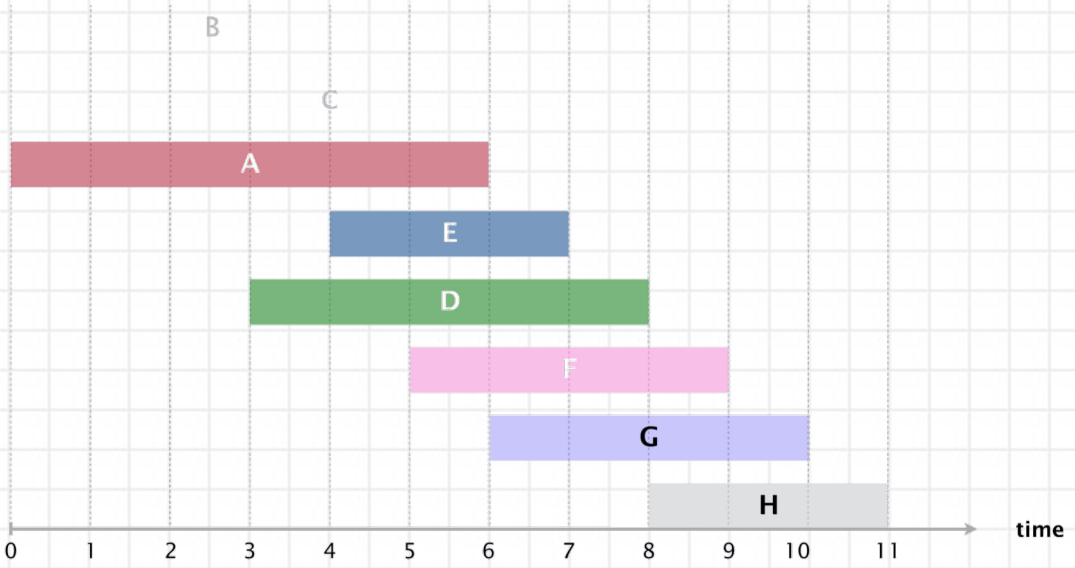
$S \leftarrow \varnothing$. ⟵ set of jobs selected

FOR $j = 1$ TO n

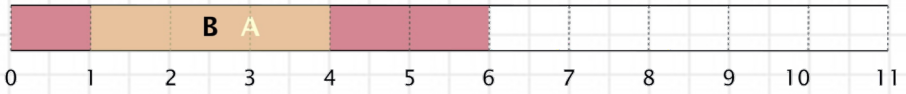
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job A is incompatible (do not add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

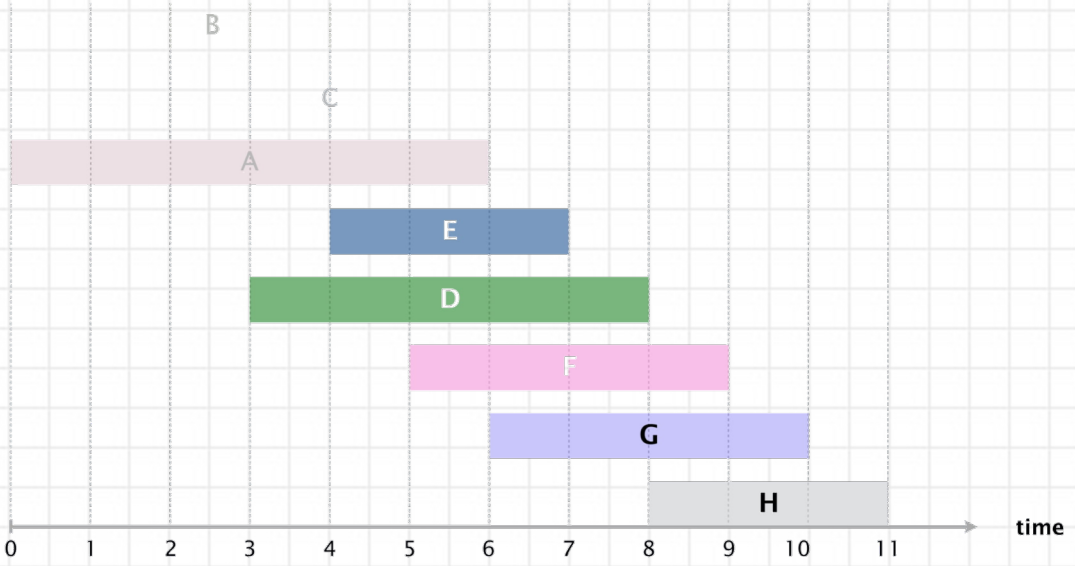
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

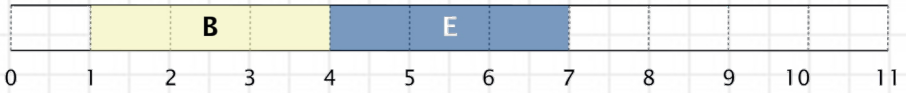
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job E is compatible (add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

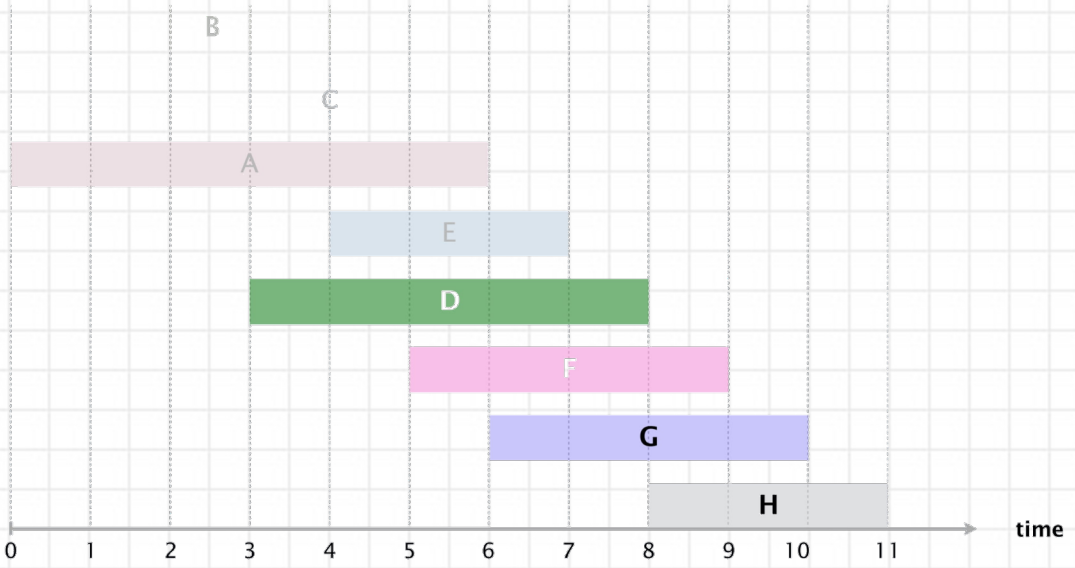
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

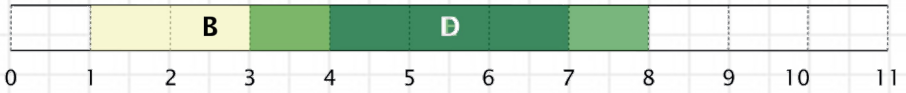
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job D is incompatible (do not add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

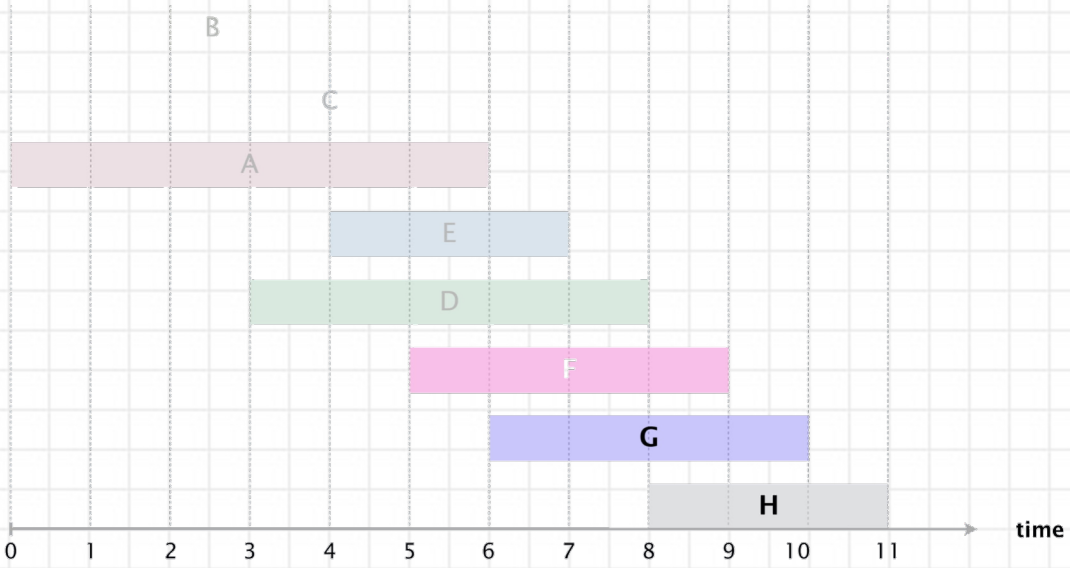
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

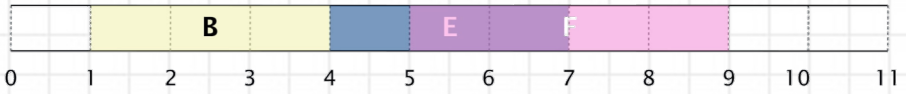
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job F is incompatible (do not add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

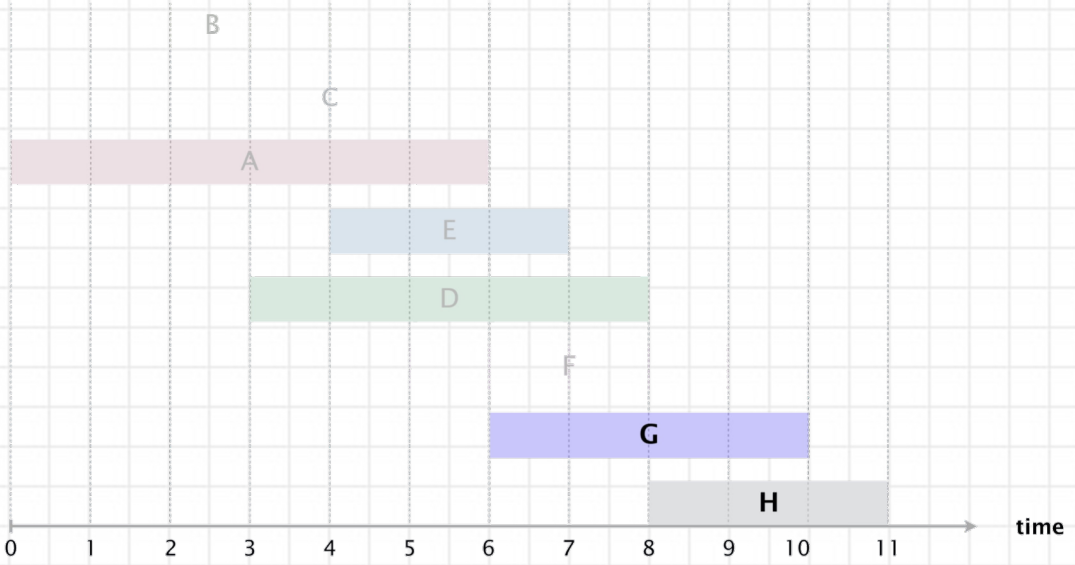
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

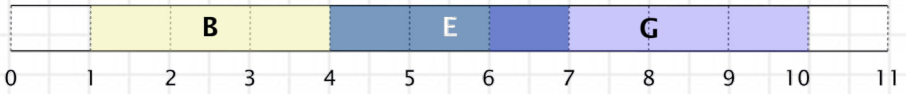
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job G is incompatible (do not add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

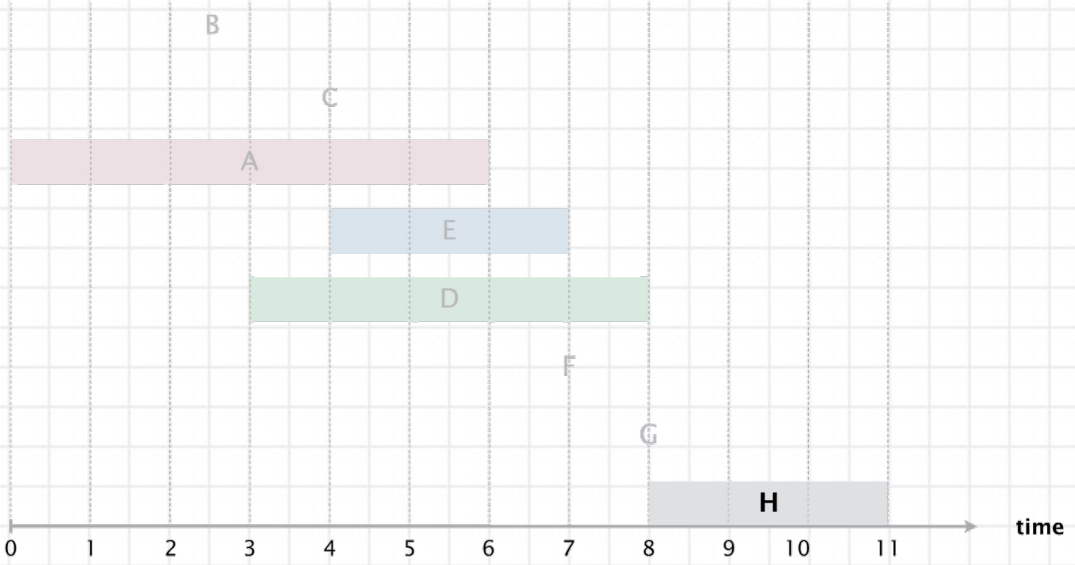
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

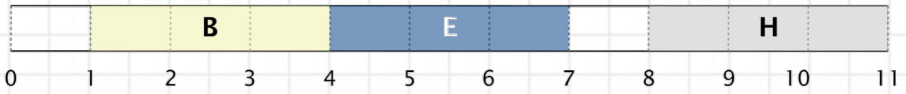
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



job H is compatible (add to schedule)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

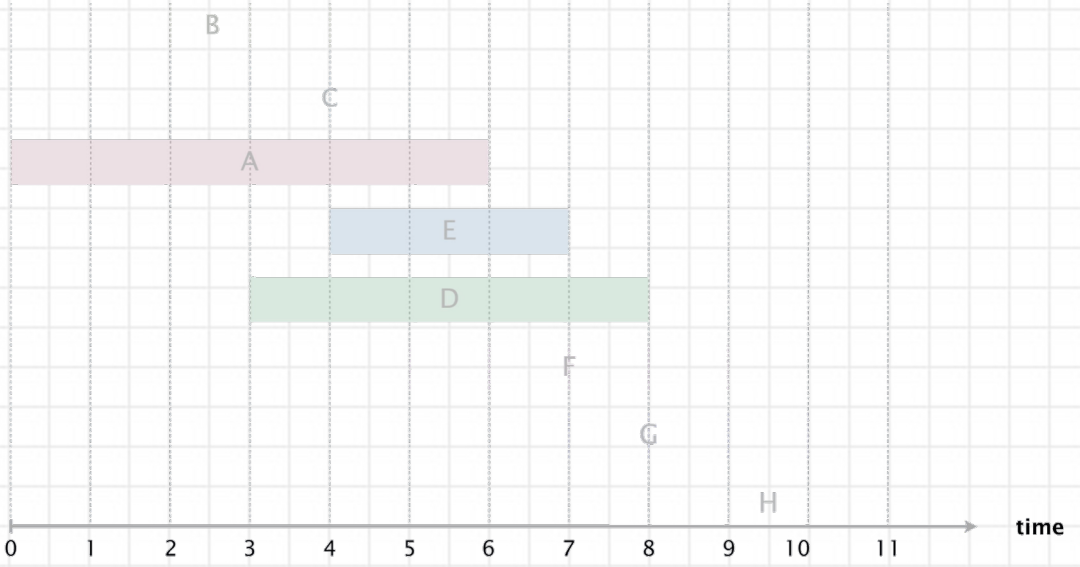
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

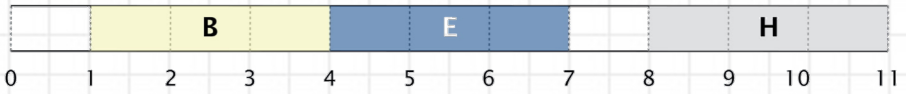
    $S \leftarrow S \cup \{ j \}$.

RETURN S.



Running time?
O(n) if sorted by finish time
O(nlogn) if it needs to be sorted first

done (an optimal set of jobs)



Greedy Example: (1) Interval Scheduling

- Greedy algorithm
  - Choose next job to add to solution as the one with earliest finish time that it is compatible with the ones already taken.
  - Natural order = finish time

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

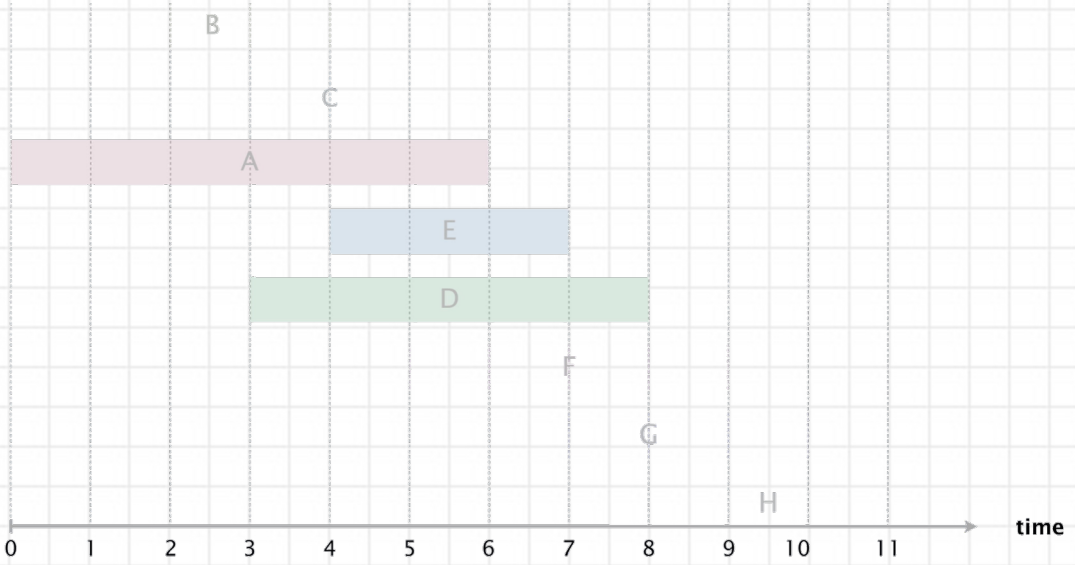
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

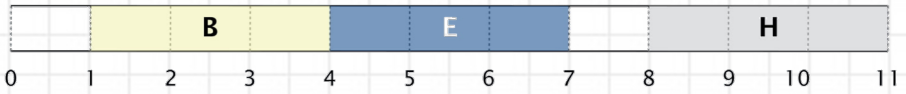
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



done (an optimal set of jobs)



But why this is optimal?

Greedy Example: (1) Interval Scheduling

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

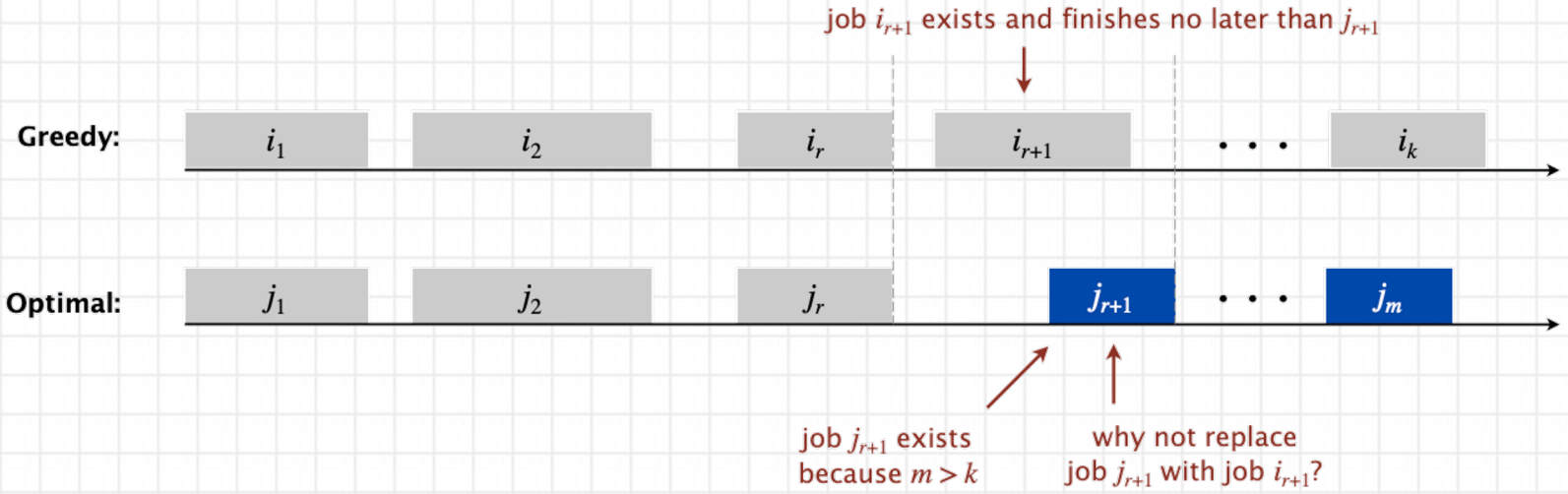
$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



Greedy Example: (1) Interval Scheduling

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

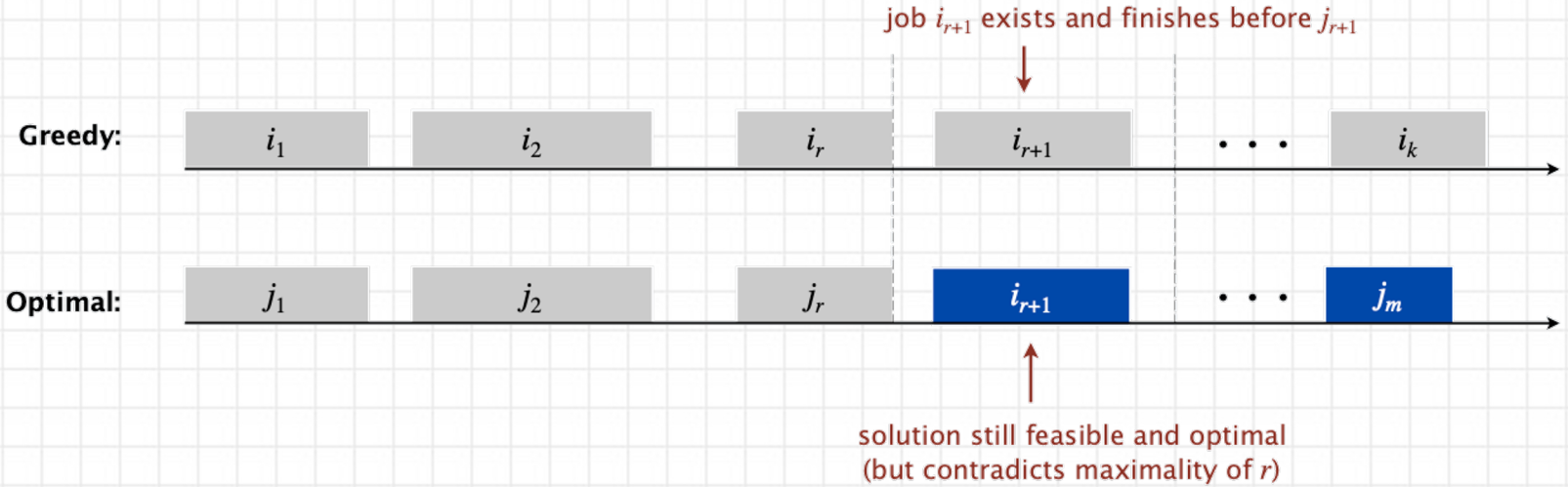
$S \leftarrow \emptyset$. ← set of jobs selected

FOR $j = 1$ TO n

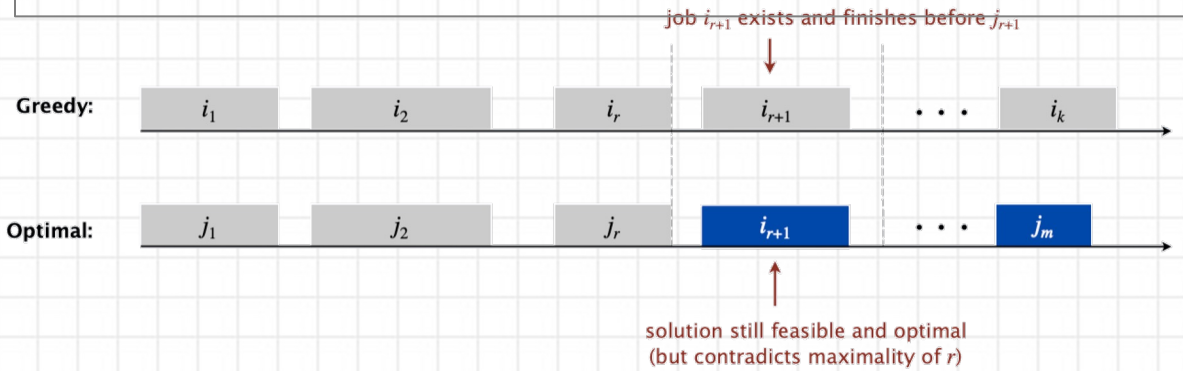
  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.



Greedy Example: (1) Interval Scheduling



- In our greedy approach, we never rule out an optimal solution.
- We build the solution step-by-step by extending the last step.
- At the end of the algorithm, we obtain some solution. Thus, it must be optimal.
- Proof by induction
  1. Base case: when we have no interval, there is an optimal solution extending that.
  2. Inductive hypothesis: After adding the i-th interval to the current optimal solution, there is an optimal solution that extends this current solution.
  3. Inductive step: Exchange argument

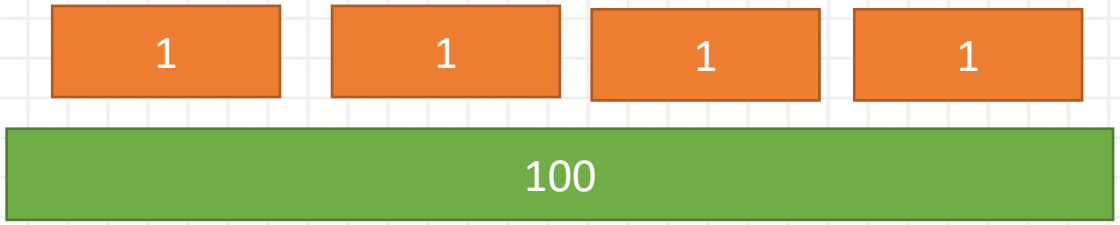
[Extra Slide] Question:

- What if each job also has a positive weight and the goal is to select a subset of mutually compatible jobs giving the maximum total weights. Does the greedy algorithm with the earliest-finish-time-first choice still work?

[Extra Slide] Question:

- What if each job also has a positive weight and the goal is to select a subset of mutually compatible jobs giving the maximum total weights. Does the greedy algorithm with the earliest-finish-time-first choice still work?

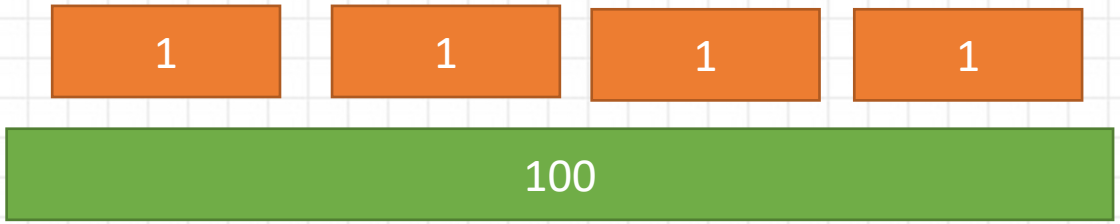
- No, counter example



[Extra Slide] Question:

- What if each job also has a positive weight and the goal is to select a subset of mutually compatible jobs giving the maximum total weights. Does the greedy algorithm with the earliest-finish-time-first choice still work?

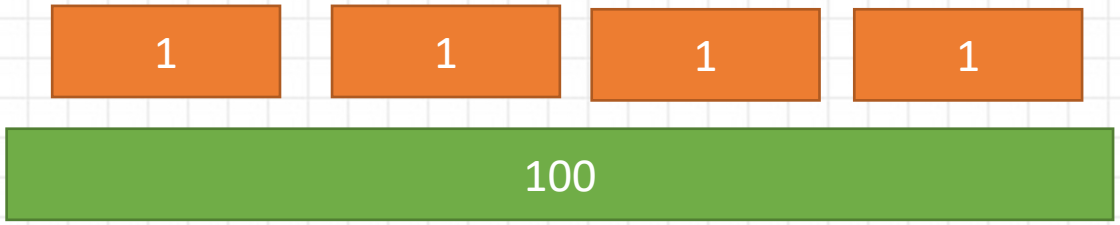
- No, counter example



- How to solve this problem?

[Extra Slide] Question:

- What if each job also has a positive weight and the goal is to select a subset of mutually compatible jobs giving the maximum total weights. Does the greedy algorithm with the earliest-finish-time-first choice still work?



- No, counter-example

- How to solve this problem?
  - Overlapping subproblems
  - Dynamic programming

[Extra Slide] Question:

- What if each job also has a positive weight and the goal is to select a subset of mutually compatible jobs giving the maximum total weights. Does the greedy algorithm with the earliest-finish-time-first choice still work?
- How to solve this problem?
  - Overlapping subproblems
  - Dynamic programming

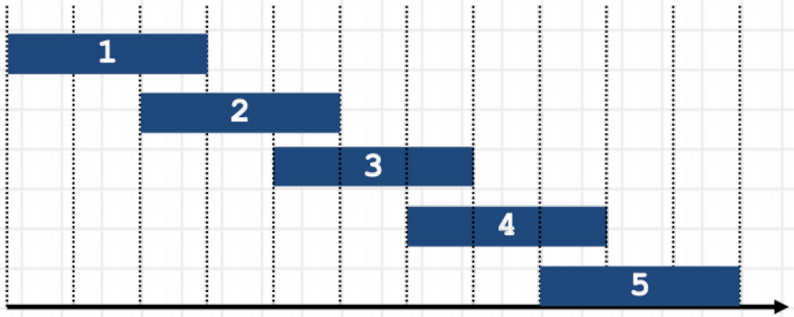
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

p(j) = largest index i < j s.t. job i is compatible with j

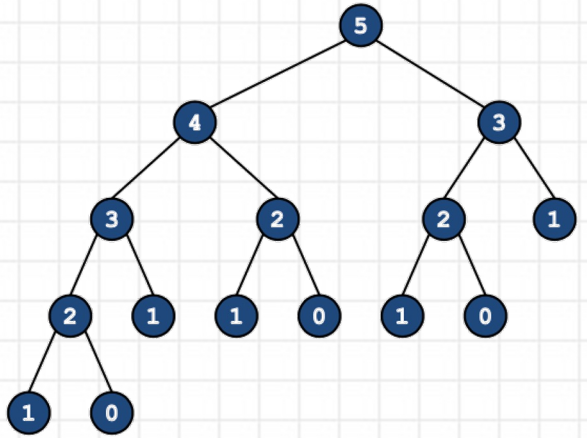
[Extra Slide] Question:

- How to solve this problem?
  - Overlapping subproblems
  - Dynamic programming
  - Ex. If each job is incompatible with only one earlier job, i.e., p(j) = j-2, then T(n) = T(n-1) + T(n-2) + O(1) → grows like Fibonacci sequence

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



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



[Extra Slide] Question:

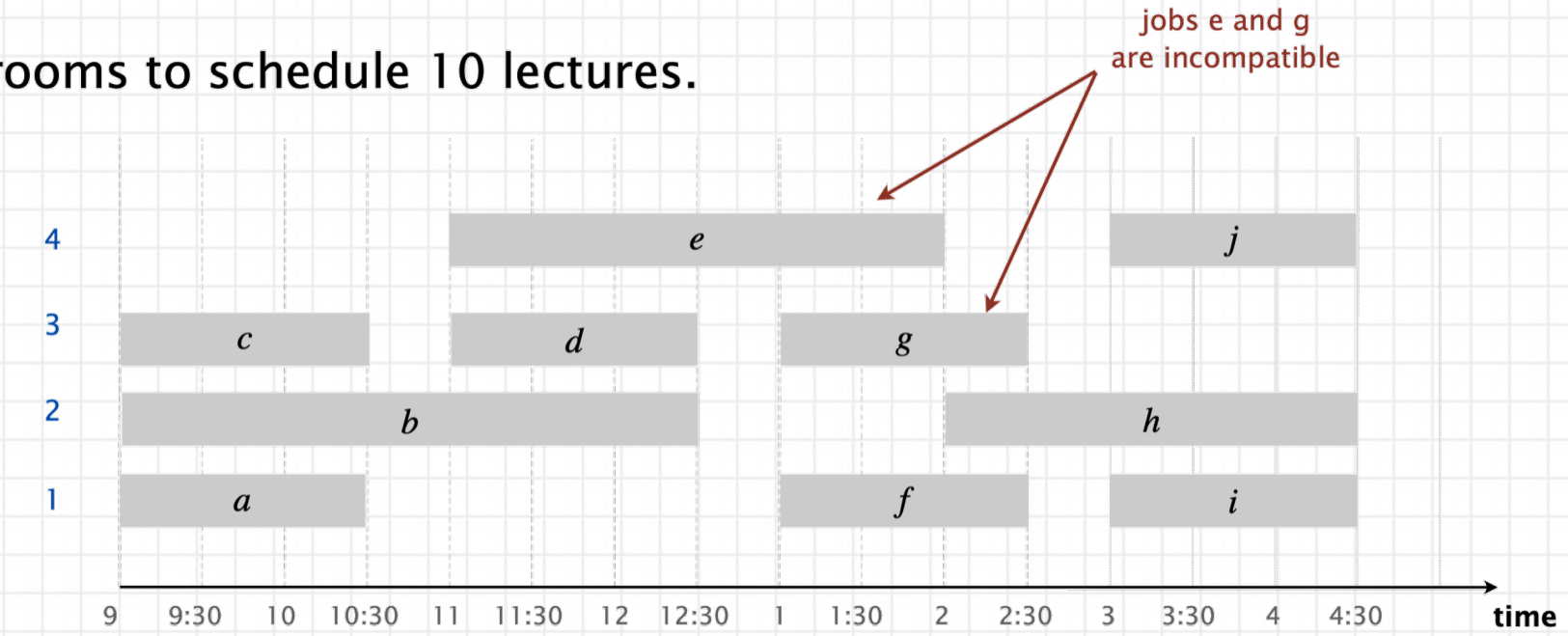
- What if each job also has a positive weight and the goal is to select a subset of mutually compatible jobs giving the maximum total weights. Does the greedy algorithm with the earliest-finish-time-first choice still work?

- No → We need dynamic programming

Greedy Example: (2) Interval Partitioning

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

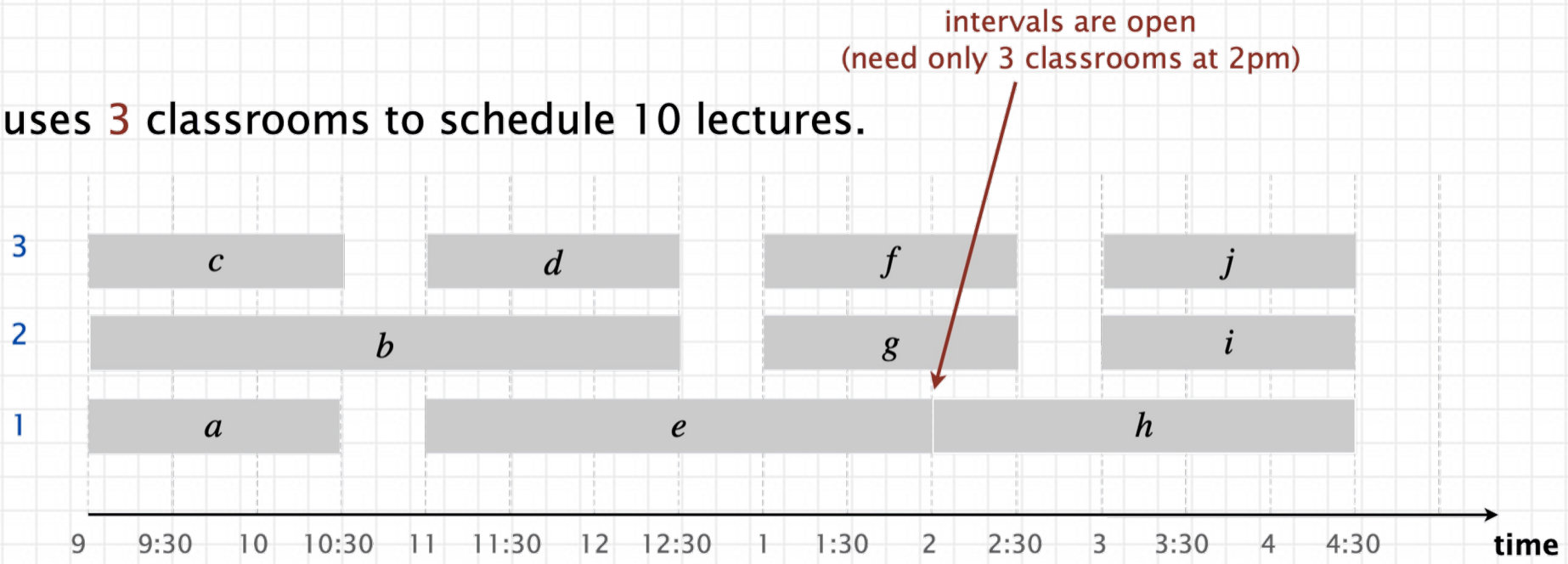
Ex. This schedule uses 4 classrooms to schedule 10 lectures.



Greedy Example: (2) Interval Partitioning

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures.

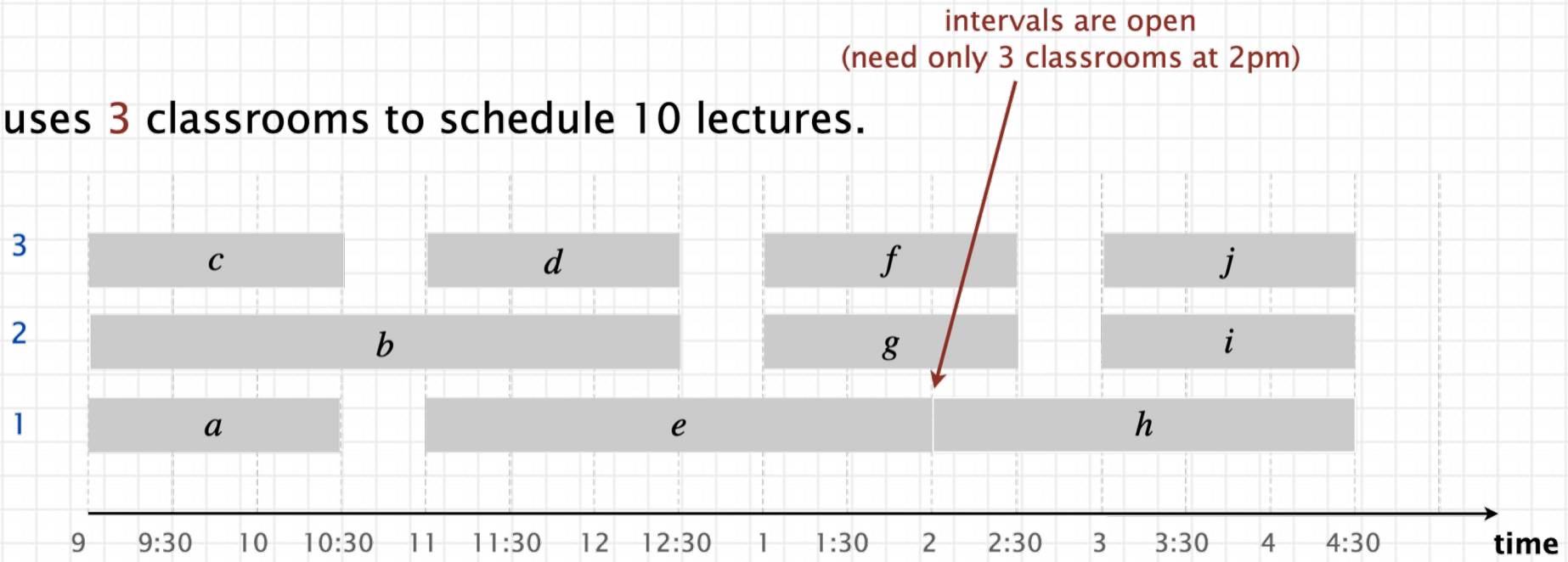


Greedy Example: (2) Interval Partitioning

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Greedy choice?

Ex. This schedule uses 3 classrooms to schedule 10 lectures.



Greedy Example: (2) Interval Partitioning

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

earliest-start-time-first

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$. ← number of allocated classrooms

FOR $j = 1$ TO n

  IF (lecture j is compatible with some classroom)

    Schedule lecture j in any such classroom k.

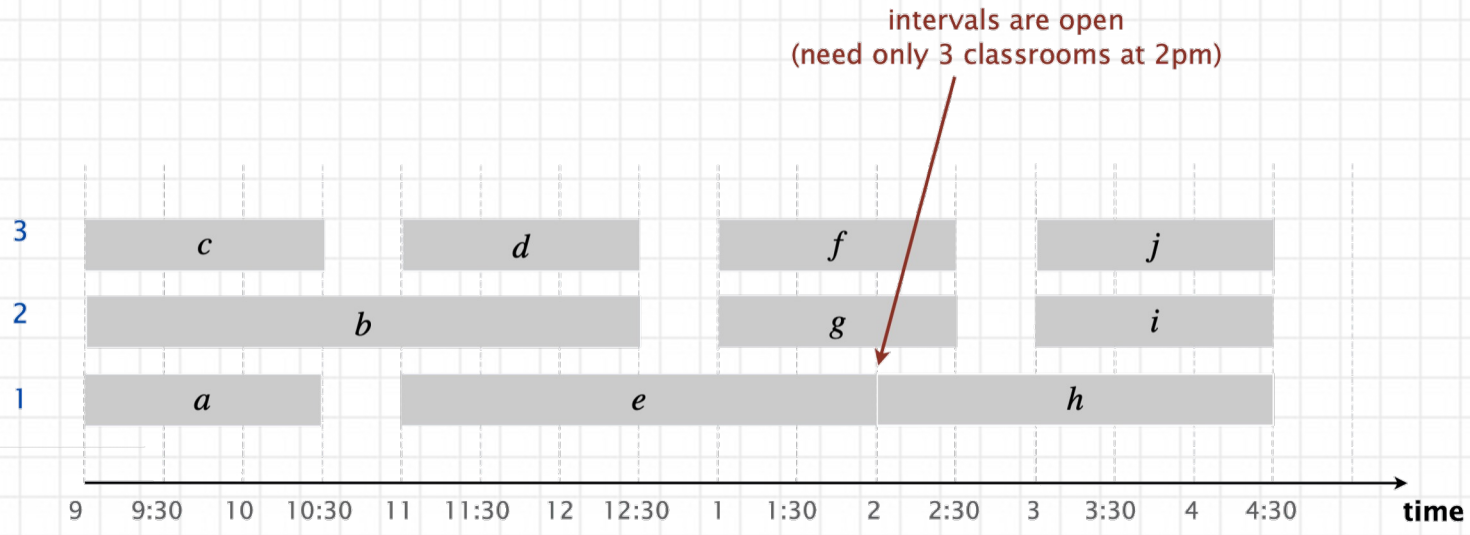
  ELSE

    Allocate a new classroom $d + 1$.

    Schedule lecture j in classroom $d + 1$.

    $d \leftarrow d + 1$.

RETURN schedule.



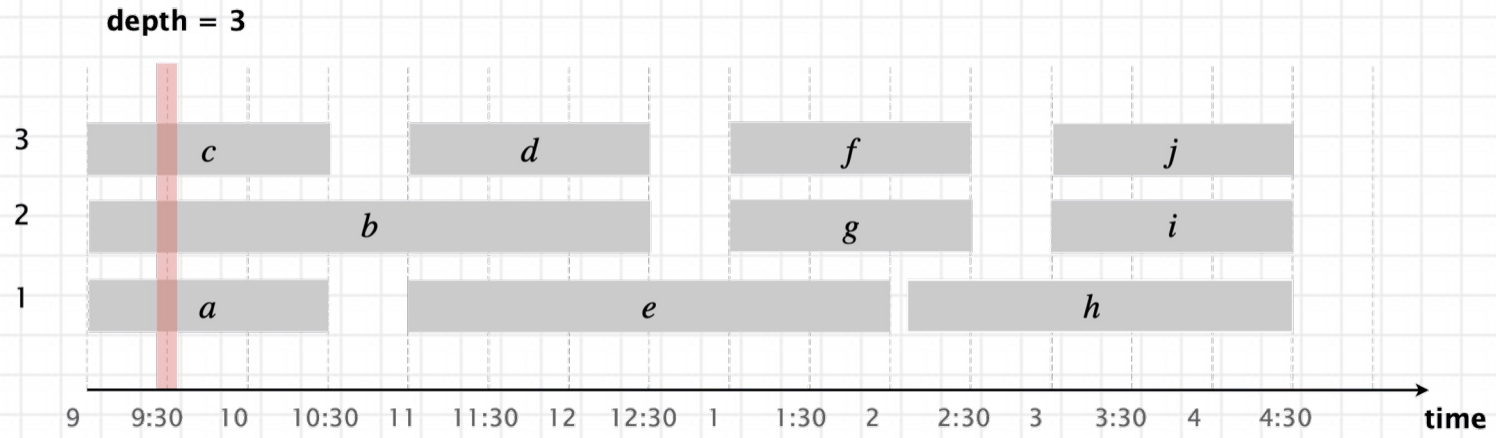
Greedy Example: (2) Interval Partitioning

Def. The depth of a set of open intervals is the maximum number of intervals that contain any given point.

Key observation. Number of classrooms needed ≥ depth.

Q. Does minimum number of classrooms needed always equal depth?

A. Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.



Greedy Example: (2) Interval Partitioning

Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest-start-time-first algorithm is optimal.

Pf.

- Let d = number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with a lecture in each of $d - 1$ other classrooms.
- Thus, these d lectures each end after s_j.
- Since we sorted by start time, each of these incompatible lectures start no later than s_j.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq d$ classrooms. ▪

Greedy Algorithms Summary

- Build the solution step-by-step
- At each step, make a decision that is locally optimal
- Needs an especially nice optimal substructure

- Applications in Graph (next week)
  - Kruskal's algorithm (minimum spanning tree)
  - Prim's algorithm (minimum spanning tree)
  - Dijkstra's algorithm (shortest path)

References

- The lecture slides are heavily based on the suggested textbooks and the corresponding published lecture notes:

  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
  - DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.
  - BRV: Benoit, A., Robert, Y., & Vivien, F. (2013). *A guide to algorithm design: paradigms, methods, and complexity analysis*. CRC Press.

  - Slides by Kevin Wayne. Copyright © 2005 Pearson-Addison Wesley.

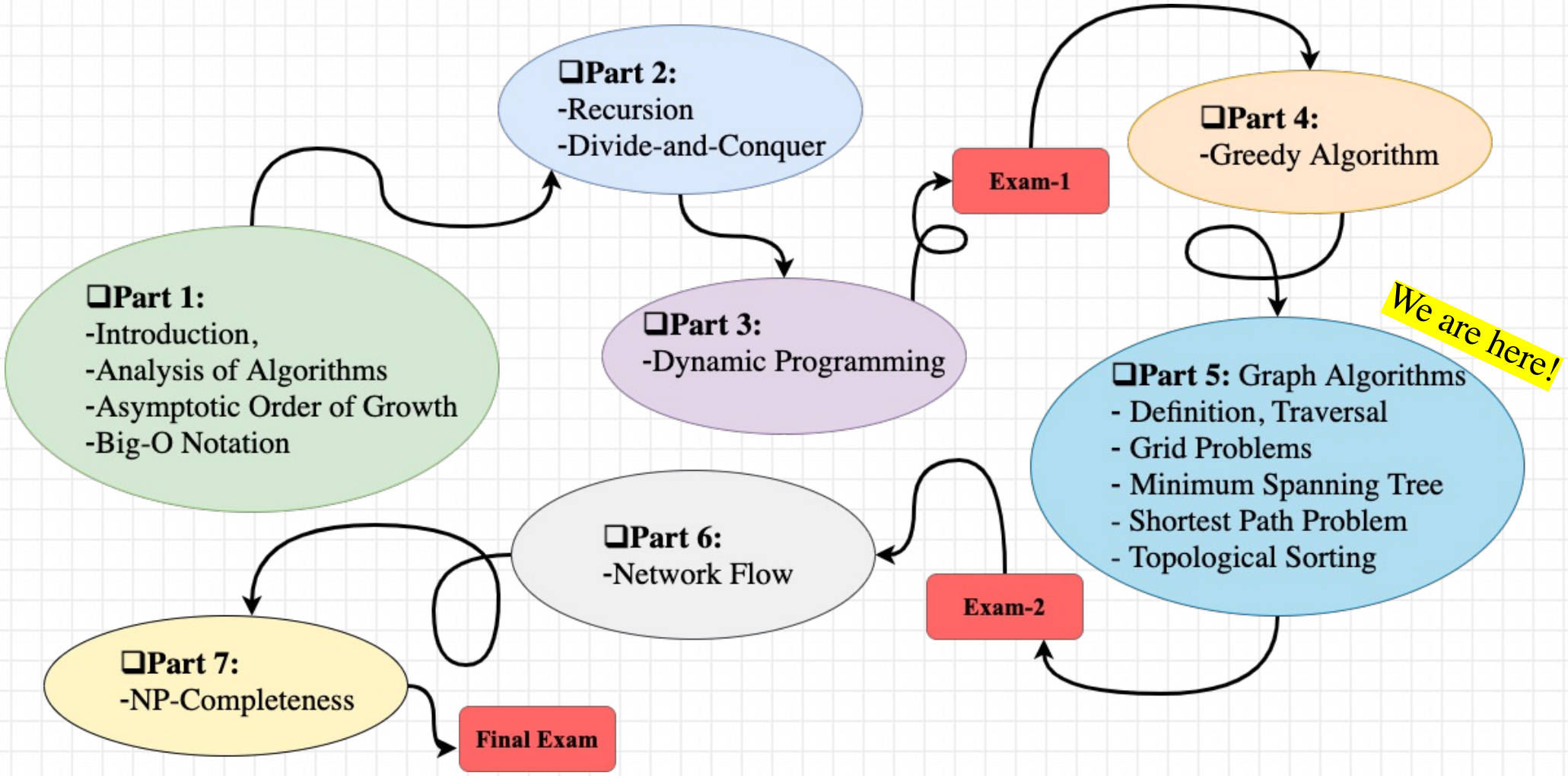
CS-3510:
Design and Analysis of Algorithms

Graph Algorithms:
Definitions and Representations

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Roadmap

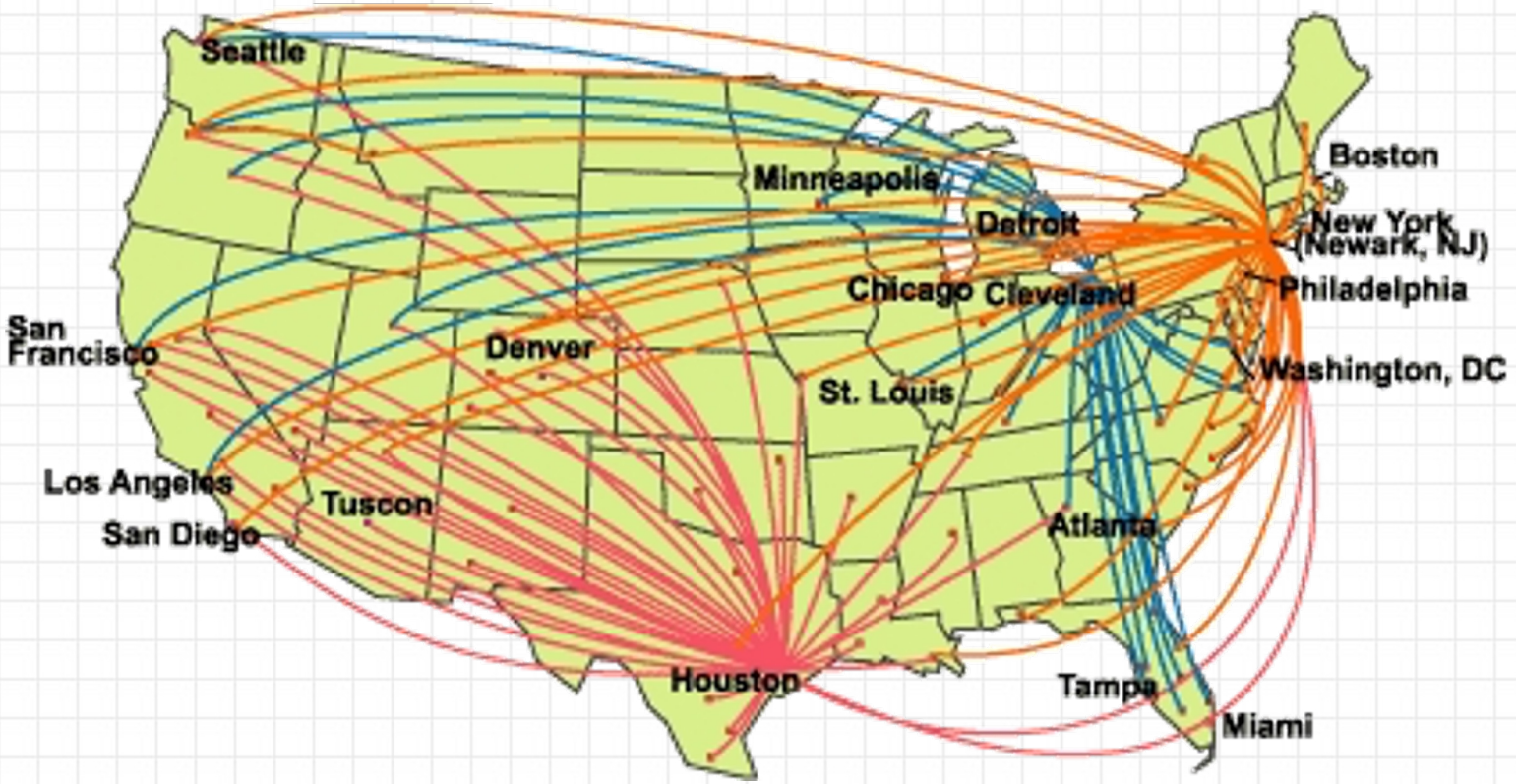

❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 2:
-Recursion
-Divide-and-Conquer
❑Part 3:
-Dynamic Programming
Exam-1
❑Part 4:
-Greedy Algorithm
We are here!
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

Graph

- Applications
- Definitions
- Matrix Representation
  - Directed Graphs
  - Undirected Graphs
  - Pros and Cons
- Linked List Representation
  - Implementation

Graph: Applications

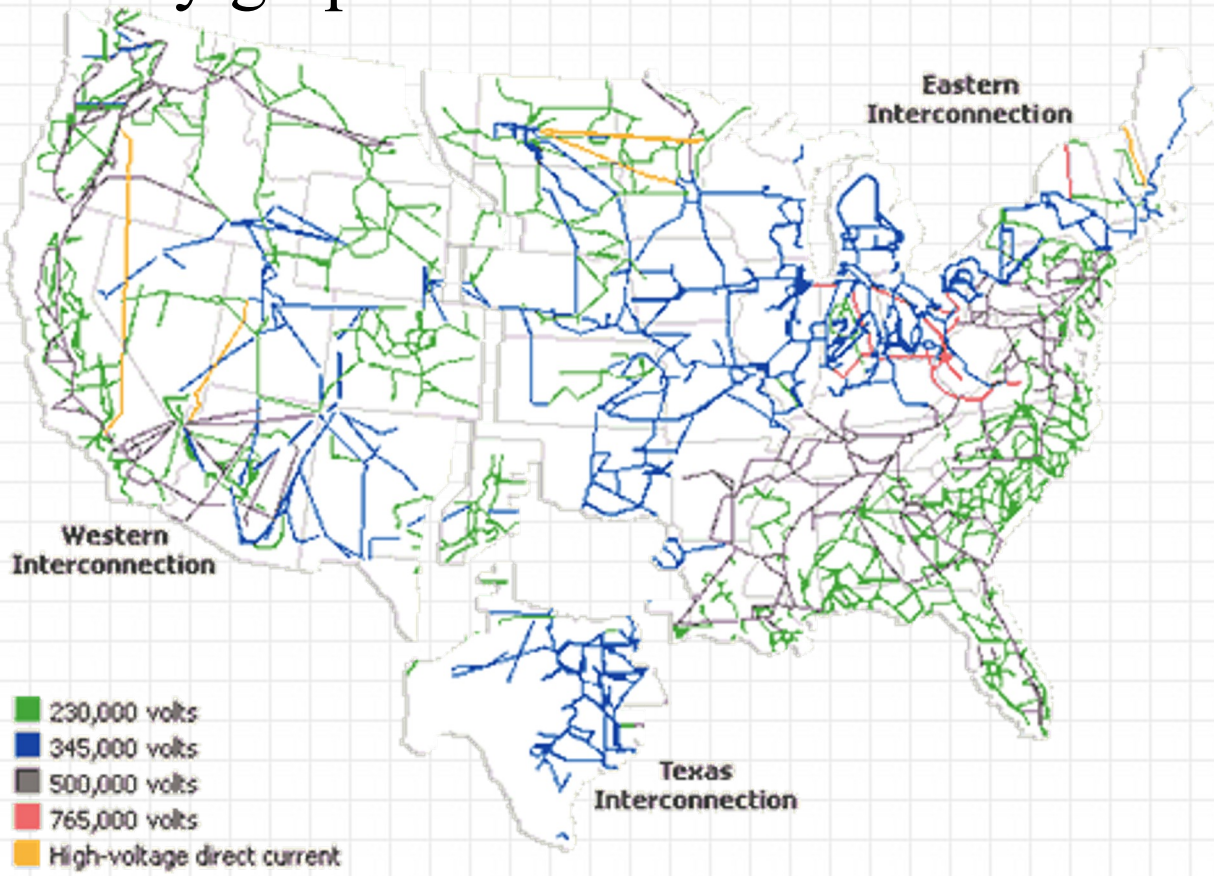
- Many problems can be represented by graphs
  - Airline routes



http://virtualskies.arc.nasa.gov/research/tutorial/images/12routemap.gif

Graph: Applications

- Many problems can be represented by graphs
  - Electric power grid

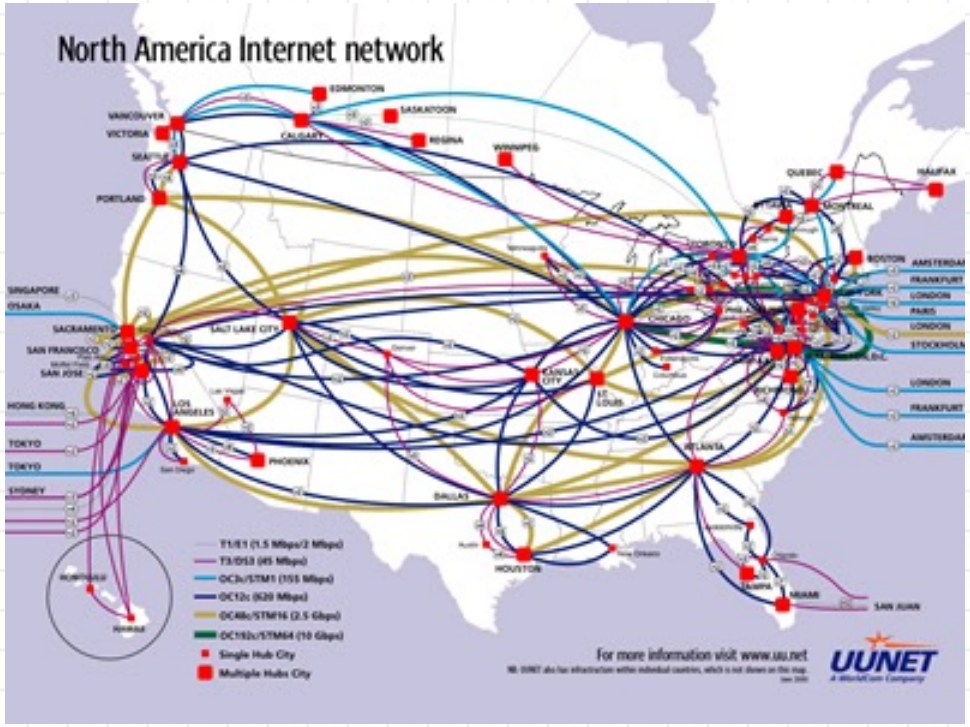


http://images.encarta.msn.com/xrefmedia/aencmed/targets/maps/map/000a5302.gif

Graph: Applications

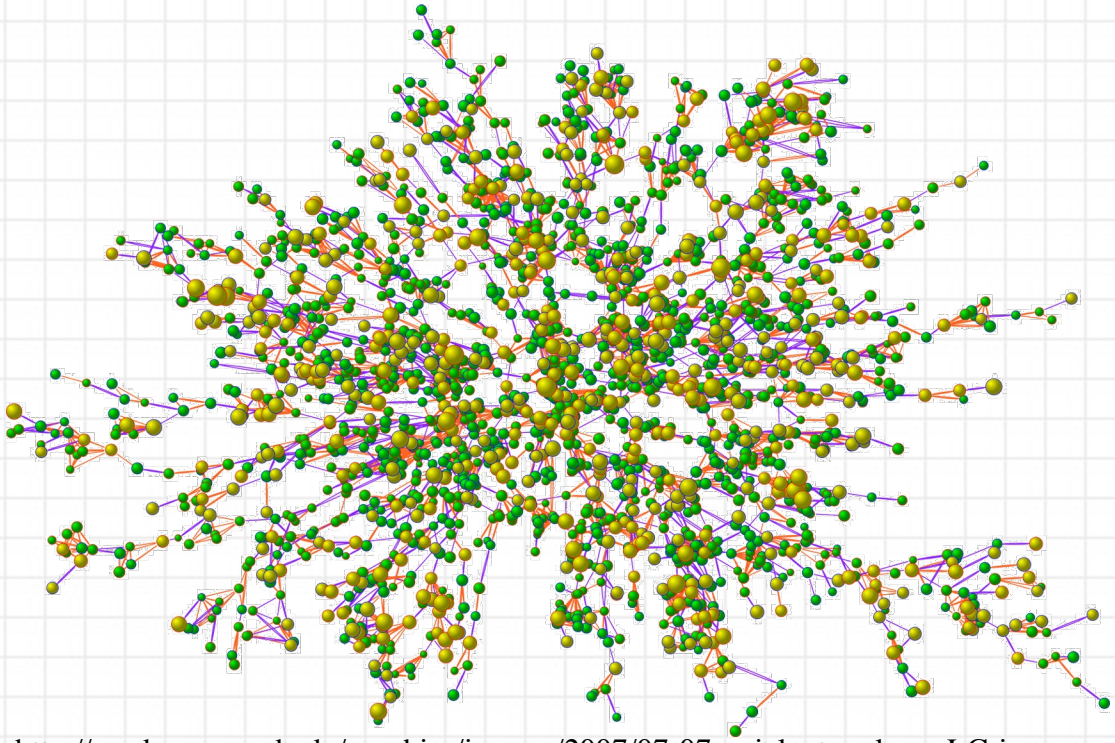
- Many problems can be represented by graphs
  - Networks

Internet



http://www.visualcomplexity.com/vc/images/270_big01.jpg

Social networks



http://ucsdnews.ucsd.edu/graphics/images/2007/07-07socialnetworkmapLG.jpg

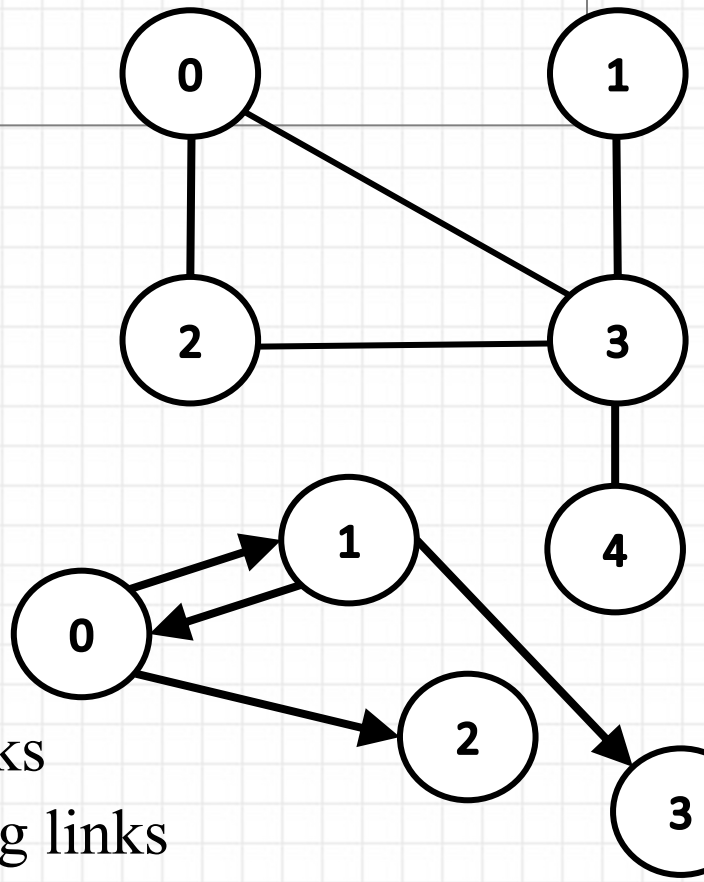
Graph: Applications

- Social networks
  - Friendships, family relationships, communications (email, phone), professional groups, physical relationships (e.g., disease spread) etc.
- Information networks
  - Citations among articles, world-wide-web (distinct from the physical communication network), preference networks (e.g., Netflix)
- Biological networks
  - Metabolic pathways, genetic regulatory networks, neural networks, food web (predator-prey relationships)
- Technological networks
  - Electric power grid, the Internet (physical computer communication network), transportation networks (airline routes, road networks, etc.), electronic circuits, …

Graph: Applications

- Routes in transportation networks
- Spread of diseases/trends/opinions
- Information propagation on the Internet (e.g., viral videos)
- Security (e.g., relationships among people)
- Cascading failures in the electric power grid
- Robustness of telecommunication networks
- Social dynamics (e.g., friendship, authorship, mentorship)
- Many others…

Graph: Definition



- Graph = (V, E)

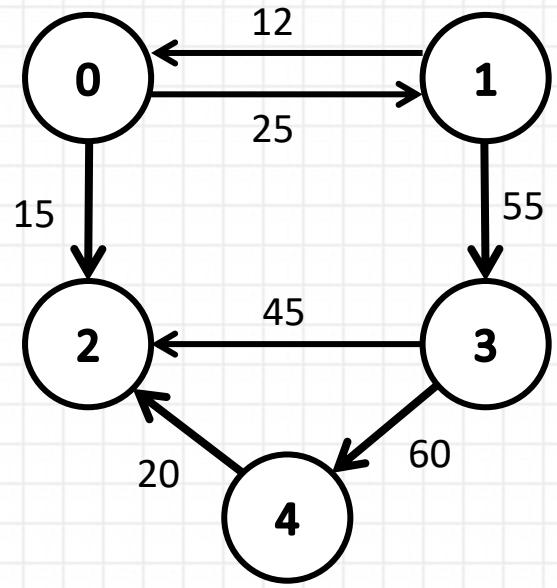
  V = Vertex (node) set = {0, 1, 2, … n-1}

  E = Edge (link) set = {e_1, e_2, … e_m} where $e_i = (v_i, v_j)$ where v_i and v_j are vertices in V

- Directed vs. undirected graphs
- Degree(i): number of edges on node i
  - In-degree (directed networks): number of incoming links
  - Out-degree (directed networks): the number of outgoing links
- Often associate properties with vertices and/or links (e.g., distance)

Representation: Adjacency Matrix

- Label nodes 0, 1, 2, … N-1
- Define matrix A[i, j]: (i, j, = 0, 1, … N-1)
  - = 1 if there is a link from node i to node j
  - = 0 if no link exists from i to j
- Alternatively, A[i,j] could represent a quantity such as the distance (or cost, or time) from node i to node j

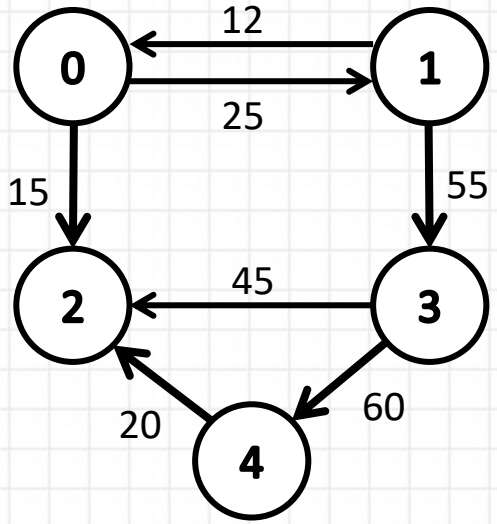


	0	1	2	3	4
0	-	25	15	-	-
1	12	-	-	55	-
2	-	-	-	-	-
3	-	-	45	-	60
4	-	-	20	-	-

The blanks actually represent 0s

Representation: Adjacency Matrix

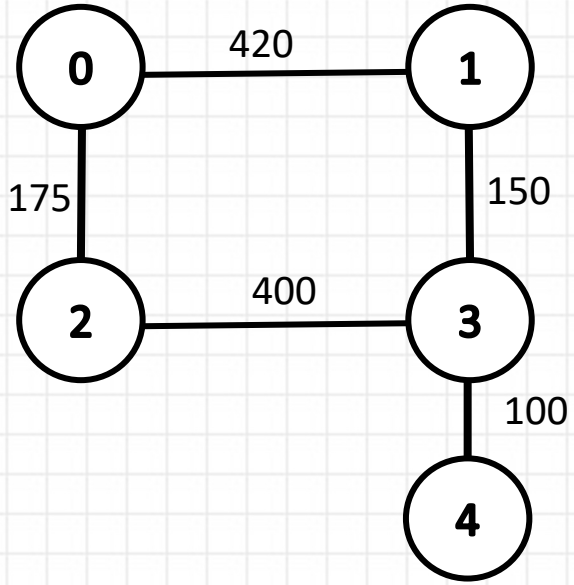
- Row i indicates the outgoing links for node i
- Column j indicates the incoming links for node j
- Here the entry indicates the length associated with that link/edge
- Length of a 2-hop path from i to j via node k can be computed as
  A[i,k] + A[k,j] where i, j, and k are distinct nodes and such a path existed
- Example: Length of path from 0 to 3 via node 1 is A[0,1]+A[1,3] = 25+55 = 80



	0	1	2	3	4
0	-	25	15	-	-
1	12	-	-	55	-
2	-	-	-	-	-
3	-	-	45	-	60
4	-	-	20	-	-

Adjacency Matrix: Undirected Graphs

- Special case of directed graphs
- For undirected graphs, each link represented by two edges, one in each direction



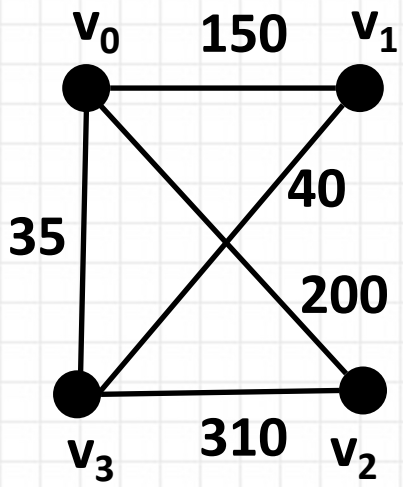
	0	1	2	3	4
0	-	420	175	-	-
1	420	-	-	150	-
2	175	-	-	400	-
3	-	150	400	-	100
4	-	-	-	100	-

- For undirected graphs, A is symmetric: $A = A^T$

Adjacency Matrix: Example

1. Draw the graph given by the adjacency matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$



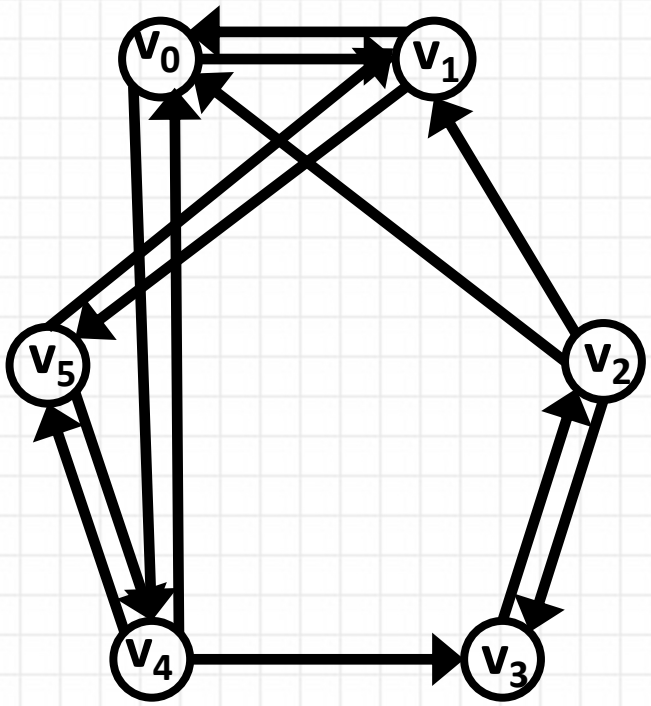
2. Use an adjacency matrix to represent the graph.

3. Is the adjacency matrix of a graph unique? Why or why not?

Adjacency Matrix: Example

1. Draw the graph given by the adjacency matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

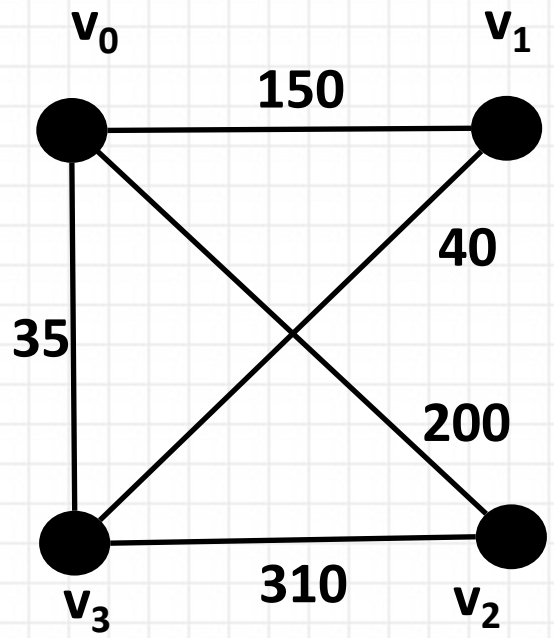


2. Use an adjacency matrix to represent the graph.

3. Is the adjacency matrix of a graph unique? Why or why not?

Adjacency Matrix: Example

2. Use an adjacency matrix to represent the graph.



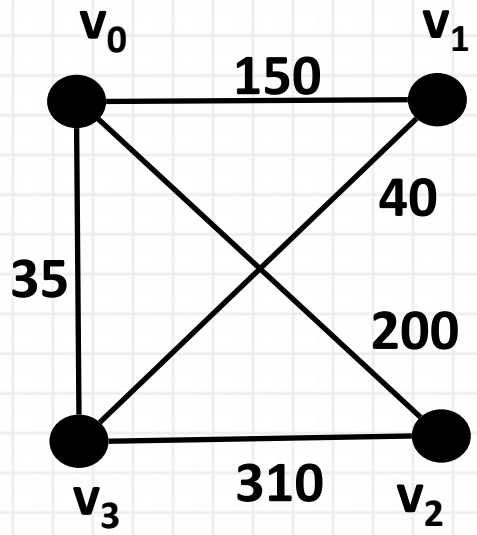
$$A = \begin{bmatrix} 0 & 150 & 200 & 35 \\ 150 & 0 & 0 & 40 \\ 200 & 0 & 0 & 310 \\ 35 & 40 & 310 & 0 \end{bmatrix}$$

3. Is the adjacency matrix of a graph unique? Why or why not?

Adjacency Matrix: Example

3. Is the adjacency matrix of a graph unique? Why or why not?

Once a vertex ordering has been specified, it is. But if we reorder the vertices, the matrix may be different. For example, if row 1 corresponded to v2 and row 2 to v1, the matrix would be different.



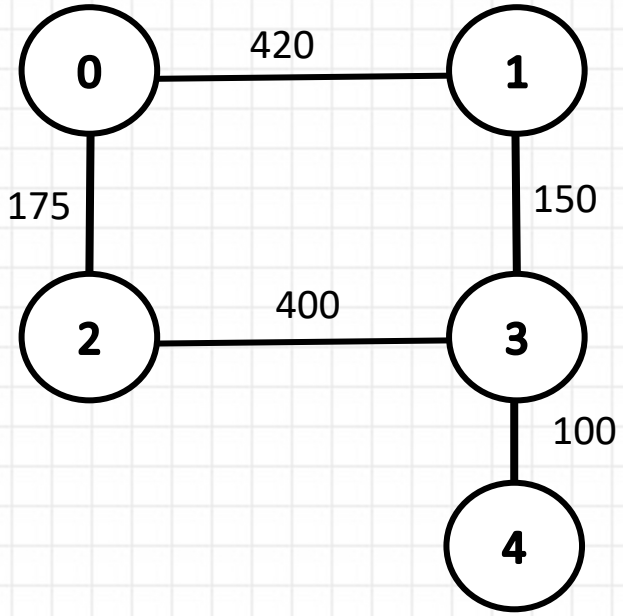
$$A = \begin{bmatrix} 0 & 150 & 200 & 35 \\ 150 & 0 & 0 & 40 \\ 200 & 0 & 0 & 310 \\ 35 & 40 & 310 & 0 \end{bmatrix}$$

Adjacency Matrix: Pros & Cons

- Easy to determine quickly if there is a link between nodes i and j
- Space = N^2 , where N = number of nodes
- Inefficient if graphs are sparse (usual case for large graphs)
  - A graph with 1000 nodes will contain 1,000,000 array elements
  - Most of these entries will be zero or "empty" for most graphs that arise in practice (although there are ways to improve efficiency for such cases)
- For undirected graphs, information is duplicated

Representation: Linked List

- Need a list of vertices; could be stored in a one-dimensional array
- For each vertex i
  - Adj[i] is a list of vertices k where (i, k) ∈ E (neighbors of i)
  - Order of nodes in adjacency list may/may not be important



4: 3:100

3: 4:100 | 1:150 | 2:400

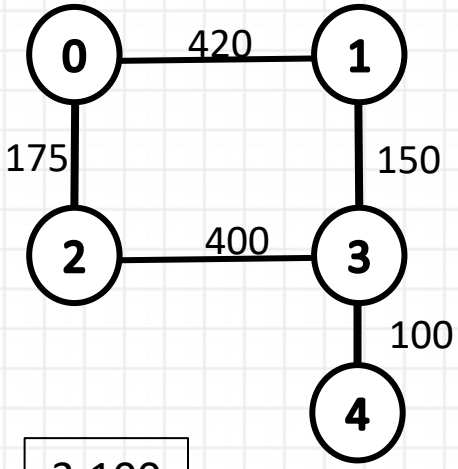
2: 3:400 | 0:175

1: 0:420 | 3:150

0: 1:420 | 2:175

Representation: Linked List

- Need a list of vertices; could be stored in a one-dimensional array
- For each vertex i
  - Adj[i] is a list of vertices k where (i, k) ∈ E (neighbors of i)
  - Order of nodes in adjacency list may/may not be important



4: 3:100

3: 4:100 | 1:150 | 2:400

2: 3:400 | 0:175

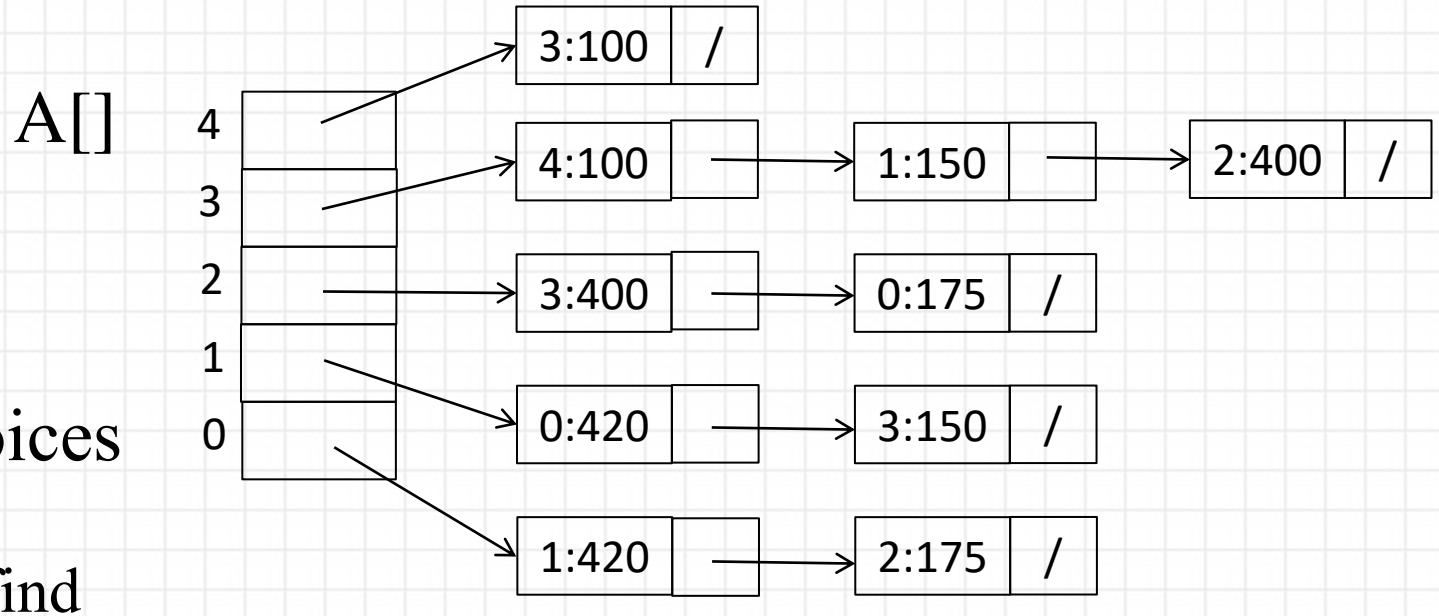
1: 0:420 | 3:150

0: 1:420 | 2:175

- Space?
  - Amount of space is 2*E + N (E edges and N vertices)
  - Preferred representation for sparse graphs
  - Some effort to determine if there is a link between two vertices

Representation: Linked List

- Store each list of edges as a linked list
- A[i] is a pointer to the list of edges to which node i connects

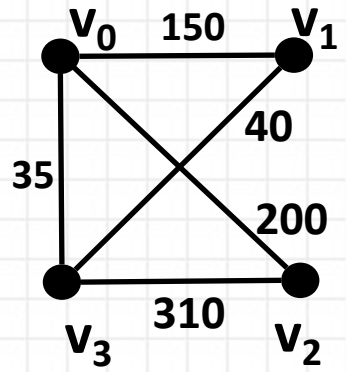


- Implementation issues/choices
  - Sorted vs. unsorted list
  - Operations: insert, delete, find
  - Dynamic graphs that change in size (nodes, edges)

Adjacency List: Example

1. Draw the graph given by the adjacency list.

4:	2:20	
3:	4:60	2:45
2:	NIL	
1:	0:12	3:55
0:	1:25	2:15



2. Use an adjacency list to represent the graph.

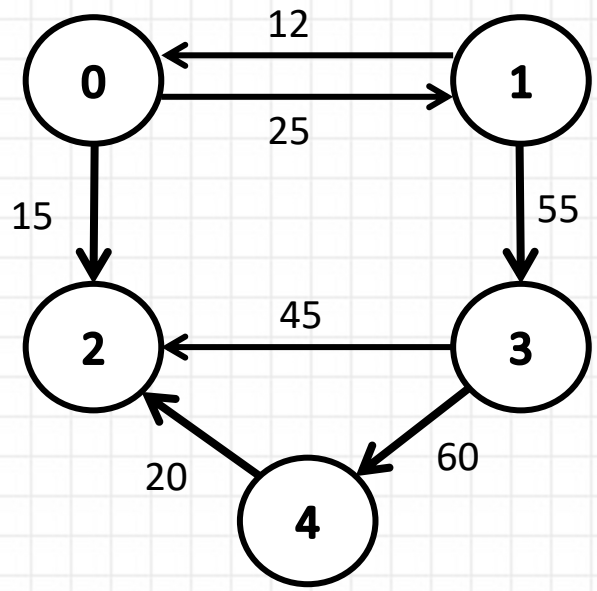
3. How can edge weights be represented with adjacency matrices/adjacency lists? How about vertex weights?

4. What are some other ways you can think of to represent graphs?

Adjacency List: Example

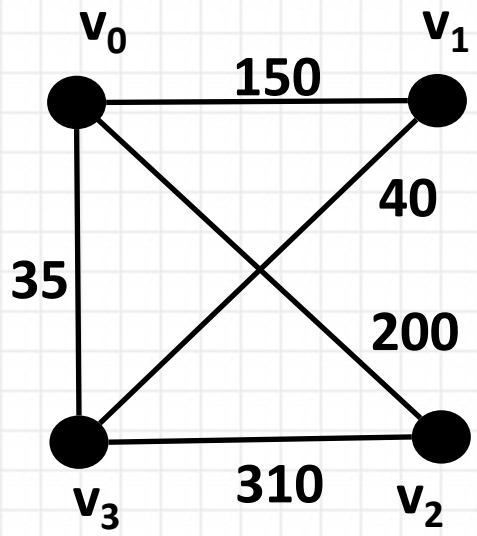
1. Draw the graph given by the adjacency list.

4:	2:20	
3:	4:60	2:45
2:	NIL	
1:	0:12	3:55
0:	1:25	2:15



Adjacency List: Example

2. Use an adjacency list to represent the graph.



3:	0:35	2:310	1:40
2:	0:200	3:310	
1:	0:150	3:40	
0:	3:35	1:150	2:200

Adjacency List: Example

3. How can edge weights be represented with adjacency matrices/adjacency lists? How about vertex weights?
   - Edge weights: With an adjacency matrix, edge weights can be the matrix entries. With an adjacency list, edge weights can be stored with each neighbor node.
   - Vertex weights: In either case, vertex weights may be best stored with the list of vertices. This is easy for an adjacency list in particular. Note that vertex weights cannot be included explicitly in an adjacency matrix.
4. What are some other ways you can think of to represent graphs?
   - One idea: build from sparse matrix representations by using a list of ordered pairs to represent edges (works for directed or undirected).
   - Another option: an incidence matrix, with each vertex represented by a row and each edge represented by a column, with nonzero entries for the two vertices associated with each edge in that edge's column.

Graph: Summary

- Graphs arise in many application areas
- Two common ways to represent graphs
  - Adjacency matrix
  - Adjacency list (a type of sparse matrix)
  - Both can be used to represent directed graphs
- Adjacency matrix
  - Space: N^2 elements for N vertices
  - Easy to check if a link exists between two vertices
- Adjacency list
  - More common representation: most large real-world graphs are sparse
  - Space: Number of edges [2*(number of edges) if undirected] + number of vertices
  - Linked list implementation is typically used

References

- The lecture slides are heavily based on the suggested textbooks and the corresponding published lecture notes:
  - Slides by Elizabeth Cherry, Georgia Institute of Technology.
  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.