CS4102 – Algorithms

Tom Horton
horton@cs.virginia.edu

Mark Floryan
mfloryan@cs.virginia.edu

- Course Mechanics
- Course content
  - Topics from earlier classes
  - Course learning objectives
- What's the course all about? A quick tour

Course introduction

General Info

- See syllabus on course website for general information
- Pre-requisites:
  - CS2150 (with C- or better)
  - Math topics: proof by induction, proof by contradiction, exponents, logarithms, limits, simple differentiations (covered in APMA 1090 or MATH 1210 or MATH 1310)
- Teaching Assistants
  - Graduates (1):
    - TBD: Will likely have one graduate student
  - Undergraduates (~20)
    - We should have enough undergraduate support!
  - Both will hold office hours, which will start next week
    - Locations and hours TBA
  - Also, we'll use Piazza for questions and Discord for office hours
    - Post all questions about HW, topics, etc. to Piazza NOT email to instructors!

Discord

- Much of our course communication will happen on Discord.
  - Class announcements
  - Office Hours
  - General Q&A with TAs, etc. (We will use Piazza too)
    - If you want instructors to see it, use Piazza
    - If someone else might search for this later, use Piazza

- Please join ASAP!
  - https://discord.gg/GqqakWbs (expires end of first week or so)

Expectations

- Of you:
  - When asked, prepare for things in advance
  - Participate in class activities when asked
  - Act mature, professional.
  - Plan ahead.
  - Don't take advantage. Follow the Honor Code. (See the BOCM.)

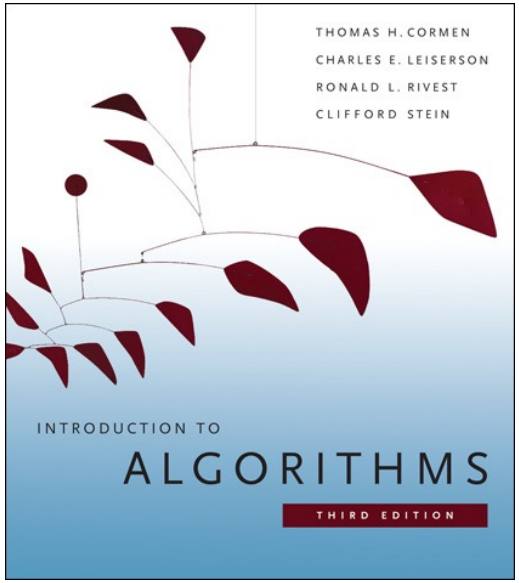
- Of me:
  - Be fair, open, and considerate.
  - Seek and listen to your feedback.
  - Not to waste your time.
  - Be effective in letting you know how you're doing

General Info

- Textbook:
  - Introduction to Algorithms, 3rd edition, by Cormen, et. al.
- Other references:
  - Your CS2150 material (will be VERY useful here)
  - Discrete Math textbook / references
- Other readings may be assigned
  - We'll see…

Textbook

- You really need to read and study material other than the slides.
- There are options, but a textbook is the easiest option.
- We'll post readings from CLRS, urge you to read them or get that info from another source.
- We may also post additional resources.



- *Introduction to Algorithms* by Cormen, et. al. (CLRS)
  - 3rd edition!
- UVA Library has a digital version of CLRS available online for free at https://search.lib.virginia.edu/catalog/u6757775

Lectures

- Back to traditional, in-person lectures.
  - 12:30 - 1:45 pm @ Olsson Hall 120 (Floryan)
  - 2:00 - 3:15 pm @ Gilmer Hall 390 (Horton)

- At least one of the two lectures (maybe both) will be recorded and posted on Collab -> Lecture Recordings
  - Using Panopto, so no live broadcast

- Lectures will cover course topics, example problems, proofs, etc.

Modules

- *The course is divided into 10 modules*
  - *Divide & Conquer: Insertion Sort, Mergesort, Quicksort*
  - *Divide & Conquer: Recurrence Relations*
  - *Divide & Conquer: Advanced Topics*
  - *Graphs: Breadth-first Search (BFS) & Depth Search (DFS)*
  - *Graphs: Kruskal's and Find-Union*
  - *Graphs: Prim's & Dijkstra's*
  - *Greedy Algorithms*
  - *Dynamic Programming*
  - *Network Flow and Ford-Fulkerson*
  - *Bi-Partite Matching & Reductions*

Modules (Cont'd)

- *Most modules are 2 lectures worth of content, some are 3 or 4 lectures.*

- *Each module involves:*
  - *2-4 lectures worth of content*
  - *1* homework assignment
  - *1 quiz (assessment)*

Quizzes

- Short assessments of your knowledge in each module
  - Meant to ensure you have knowledge of the individual topics from lecture to a sufficient degree.

- There are five different dates to take quizzes (in lecture)
  - Tue, Sep. 21 — Mod 1-3 (first attempts)
  - Thu, Oct. 7 — Mod 1-3 (second attempt), 4-5 (first attempt)
  - Tue, Nov. 16 — Mod 4-5 (second attempt), 6-8 (first attempt)
  - Thu, Dec. 2 — Mod 6-8 (second attempt), 9-10 (first attempt)
  - Final Exam — Mod 1-10 (final attempts)

Note: it's naturally harder to do a lot of quizzes in a 75-minute class period, so don't get yourself into a bad situation!

Quizzes

- Three possible grade outcomes ***for each module***:
  - ***Incomplete***: Knowledge not shown
  - ***Pass***: You clearly demonstrate competence for this module
  - ***High-Pass***: You did VERY well on this quiz

- You always receive the highest grade over all attempts
- Your quiz grade for a module can never decrease
- You can take module quizzes multiple times
- You never need to retake a quiz once you get a high-pass
  - Though you may decide that "pass" is good enough for you and choose not to retake a particular quiz

Quizzes (quick example)

- Quiz Day 1:
  - Floryan takes quizzes 1-3
  - Floryan receives a high-pass, pass, and incomplete (fail) on them respectively. **Note that the score is PER MODULE

- Quiz Day 2:
  - Floryan DOES NOT need to retake module 1
  - Floryan MIGHT choose to retry module 2 (get pass to high pass)
  - Floryan will attempt modules 4-5 for the first time

- In general: always attempt quizzes you haven't passed ***FIRST***, then try to retake old quizzes you've passed to get high-pass second.

Final Exam

- Our final exam time will be used to:
  - Provide you with ***one more attempt at every quiz***
  - You will ***ONLY attempt quizzes*** for which you ***haven't passed*** or wish to increase your grade further (to high-pass)
  - This means if you've already passed every quiz, you do not need to do anything during the final exam period
    - Likewise, some of you will come in to take 1 or 2 quizzes only, and that is fine.
  - We ***DO NOT RECOMMEND*** attempting all 10 quizzes once during the final.

Homeworks

- Each of the 10 modules has one homework associated
- ***Programming HW***:
  - Can be written in Java, C++, or Python (your choice)
  - Homework will specify the problem, input, and output specs.
  - Some will be graded solely on passing test-cases and/or meeting a target runtime.
  - Some will require you to measure run-times and answer questions.
- ***Written HW***
  - Solving small problems, analyzing runtimes, etc.
  - Sometimes proofs of correctness, etc.
  - Written in Latex (tutorial on course webpage)

Homework Grades

- Homeworks are pass / fail and meant to be fairly low-stress (compared to other classes):
  - ***Incomplete***: The student has not submitted evidence that they have engaged with the material and with the assignment.
  - ***Pass***: The student has shown evidence that they have attempted the ENTIRE assignment and made a serious, thoughtful attempt at it.

- You may submit homework assignments ***as many times as you'd like*** until you pass.

Homework Grades (Cont'd)

- A ***Pass*** does NOT mean that your homework is perfect. Simply means you clearly have put in the effort we expect.

- On a programming HW, a pass might mean:
  - You are passing simple and moderate test cases but your code is still a little too slow. You "pass" the assignment, but are encouraged to continue investigating how you can more cleanly solve the problem.
- On a written homework:
  - You have made a serious, well-written attempt at every problem even if the solutions have some issues.

Homework Grade Philosophy

- Wait, so homework is all effort-based?
- Well…not quite. The purpose of the homework is:
  - To ***practice*** in an environment that is lower-stress
  - To ***push yourself*** to solve algorithms problems to prepare you for the quizzes, NOT just to get a grade.
  - To ***tinker and experiment*** with your code (e.g., what happens if I slightly change the base case on this algorithm)
  - To focus on ***attempting to solve problems yourself*** before asking others for assistance.
  - To show us that you ***engaged with the homework*** by showing that work is "mostly" there.

Quiz and Homework Deadlines

- Quiz Deadlines:
  - Are in-person during the set lecture dates (see previous slides)
  - No makeups provided unless you have extreme extenuating circumstances (e.g., Covid-19 Quarantine, etc.)
    - Remember: you get 3 attempts at Quizzes 1-8 and 2 attempts for Quizzes 9-10

- Homework Deadlines:
  - Each homework will have a recommended deadline (about 1 per week)
    - If you want to succeed, you need to try to hit these recommended deadlines
  - Deadlines are all soft deadlines (i.e., everyone gets an automatic extension until the end of the semester.
  - All homework is due Fri., Dec. 3 (except perhaps the last HW)

Grading Overview

- Let's look at how grades work on course website →

- Your goal is to ***pass modules***
- You ***pass a module*** by:
  - Passing the homework for that module AND
  - Passing the quiz (you DO NOT need a high-pass)

- Your final letter grade is determined by how many modules you pass
  - High-passing the quiz in a module can raise your grade a bit more.

Grading Scheme Philosophy

- For grades ***F-B***, passing new modules is more important than high-passing old ones
  - We care about breadth over depth until you reach 9 modules passed to earn a B
- High-pass can be used to ***raise your grade slightly*** at each grade level.
- After B obtained, high-passes are needed to earn, B+, A-, etc.
- We are treating the course as having 9 modules (for B) instead of 10, effectively allowing you to "skip one module"

Why this Approach to Grades?

It's a good match of CS department policy on course letter grades: http://ugrads.cs.virginia.edu/grading-guidelines.html

We Think These Are Benefits:

- It values competence in a breadth of modules
- Lower stress from HWs: practice, explore, tinker,…
- Lower stress because you can repeat quizzes
  - Many quizzes, more chances to take
- "Visible:" You know what you've earned so far, can predict scores on future modules to see where you might end up
- Small grading changes have correspondingly small impacts on final grade

Office Hours

- Let's discuss office hours by looking at the course website.

Homework: Programming Hints

- Understand the problem!
- Consider all boundary cases
- Use pre-existing library code
  - Number formatting: NumberFormat in Java, printf() in C/C++
  - Input: Scanner in Java, scanf() in C, cin in C++
- Know how to handle floating point numbers
  - Understand float/double precision issues
  - Rounding, floating-point mod
- Make sure it works for the provided test cases
- Then write some of your own
- Make sure you read the language specific details for submission!!!

Homework: Programming FAQ

- Do I need to write my own sorting methods.
  - No, unless the point of that assignment is to write sorting methods, you can use libraries for this.

- Can I get the test cases from submission server?
  - No, part of the point is to work on brainstorming cases your code is missing without being told. Submission server will give limited feedback on purpose!

- Will you help me debug my code?
  - No, we won't. You need to learn how to do this on your own. I'm happy to give you advice on how to approach your debugging problems, but I will not sit down and debug code with you.

Homework: Written

- These assignments must be typeset in LaTeX
- I will provide a couple tutorials, guides, and templates when the first assignment is given out
- You may not embed images of text or formulas!

Use of Online Code Etc.

- Studying code online is permitted but only for getting ideas
- Copying or reusing code from an online source violates the pledge
  - You must cite sources of any online code you use in this way in a comment in your source file(s)

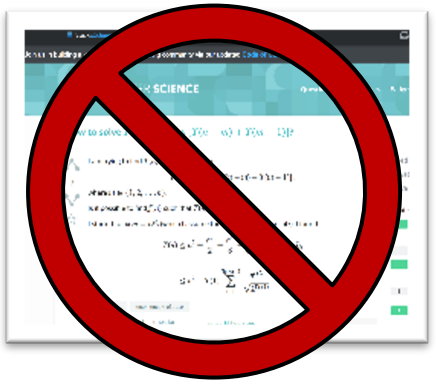
- Remember: the purpose of the homework is
  - To ***practice*** in an environment that is lower-stress
  - To ***push yourself*** to solve algorithms problems to prepare you for the quizzes, NOT just to get a grade.
  - To ***tinker and experiment*** with your code (e.g., what happens if I slightly change the base case on this algorithm)
  - To focus on ***attempting to solve problems yourself*** before asking others for assistance.

Working in groups

- For the homeworks, you may work together in groups of 3 or less to discuss the algorithmic aspects ONLY
  - State who you worked with (in code comments)
  - Do not look at or copy another student's code!
- For the written homeworks, you may work together in groups of 3 or less, but you MUST:
  - State who you worked with
  - Type up your own assignment!

Academic Integrity

- Collaboration allowed and encouraged!
  - But only within your groups of up to 3 per assignment (you + 2 more)
- Write-ups/code written independently
  - DO NOT share written solutions / code
  - DO NOT share documents (ex: Overleaf)
  - DO NOT share debugging of code
- DO NOT seek published solutions online
- Be able to explain any solution you submit!
- We will find those who don't do the right thing.



What you know already from CS2150

- Definition of an algorithm
- Definition of algorithm "complexity"
- Measuring worst-case complexity
- Cost as a function of input size
- Asymptotic rate of growth: Big-Oh, Big-Theta
- Relative ordering of rates of growth
- Analyzing an algorithm's cost:
  - sequences, loops, if/else, functions, recursion
- Focus on counting one particular statement or operation; don't count all statements

What you know already from CS2150

- Problems and their solutions:
  - Linear data structures vs. tree data structures
  - Searching: linear/sequential search, binary search (?), hashing
  - Sorting: quicksort, mergesort in CS2110 (?)
  - Priority Queue ADT and Heap Implementation

What you know already from all your courses

- Examples of Algorithm design methods:
  - Divide and Conquer (quicksort, mergesort)
  - Greedy (though you didn't call it this)
  - Dynamic programming (fibonacci numbers, Floyd-Warshall)
  - NP-complete (traveling salesperson. Have you seen this?)

What you know already from Discrete Math and Theory of Computation…

- From CS2102:
  - Proofs: induction, contradiction
    - If you are uncomfortable with these two, review them NOW!
  - Counting, probability, combinatorics, permutations

- From some earlier math class:
  - Exponents, logarithms, limits, differentiation on polynomials and other simple functions

- From CS3102 (if you have taken it)
  - Maturity in mathematics and computing theory
  - Ability to do proofs
  - Abstract models of computation, such as Turing machines

Questions? Concerns? Wrath to vent?

A first algorithm: making change

OK… But What's It Really All About?

- Let's illustrate some ideas you'll see throughout the course
  - Using one example
- Concepts:
  - Describing an algorithm
  - Measuring algorithm efficiency
  - Families or types of problems
  - Algorithm design strategies
    - Alternative strategies
  - Lower bounds and optimal algorithms
  - Problems that seem very hard

Everyone Already Knows Many Algorithms!

- Worked retail? You know how to make change!
- Example:
  - My item costs $4.37. I give you a five dollar bill. What do you give me in change?
  - Answer: two quarters, a dime, three pennies
  - Why? How do we figure that out?

Making Change

- The problem:
  - Give back the right amount of change, and…
  - Return the fewest number of coins!
- Inputs: the dollar-amount to return
  - Also, the set of possible coins. (Do we have half-dollars? That affects the answer we give.)
- Output: a set of coins

- Note this problem statement is simply a transformation
  - Given input, generate output with certain properties
  - No statement about how to do it.
- Can you describe the algorithm you use?

A Change Algorithm

1. Consider the largest coin
2. How many go into the amount left?
3. Add that many of that coin to the output
4. Subtract the amount for those coins from the amount left to return
5. If the amount left is zero, done!
6. If not, consider next largest coin, and go back to Step 2

Is this a "good" algorithm?

- What makes an algorithm "good"?
  - Good time *complexity*. (Maybe space complexity.)
  - Better than any other algorithm
  - Easy to understand
- How could we measure how much work an algorithm does?
  - Code it and time it. Issues?
  - Count how many "instructions" it does before implementing it
  - Computer scientists count basic operations, and use a rough measure of this: order class, e.g. O(n lg n)

Evaluating Our Greedy Algorithm

- How much work does it do?
  - Say C is the amount of change, and N is the number of coins in our coin-set
  - Loop at most N times, and inside the loop we do:
    - A division
    - Add something to the output list
    - A subtraction, and a test
  - We say this is O(N), or linear in terms of the size of the coin-set
- Could we do better?
  - Is this an *optimal algorithm*?
  - We need to do a proof somehow to show this

You're Being Greedy!

- This algorithm is an example of a family of algorithms called *greedy algorithms*
- Suitable for optimization problems
  - There are many *feasible answers* that add up to the right amount, but one is optimal or best (fewest coins)
- Immediately greedy: at each step, choose what looks best now. No "look-ahead" into the future!

- What's an optimization problem?
  - Some subset or combination of values satisfies problem constraints (feasible solutions)
  - But, a way of comparing these. One is best: the optimal solution

Does Greed Pay Off?

- Greedy algorithms are often efficient.
- Are they always right? Always find the optimal answer?
  - For some problems.
  - Not for checkers or chess!
  - Always for coin-changing problem? Depends on coin values
    - Say we had a 11-cent coin
    - What happens if we need to return 15 cents?
  - So how do we know?
- In the real world:
  - Many optimization problems
  - Many good greedy solutions to some of these

Formal algorithmic description

- *All* algorithms in this course must have the following components:
  - Problem description (1 line max)
  - Inputs
  - Outputs
  - Assumptions
  - Strategy overview
    - 1 or 2 sentences outlining the basic strategy, including the name of the method you are going to use for the algorithm
  - Algorithm description
    - If listed in English (as opposed to pseudo-code), then it should be listed in steps

Change solution (greedy)

- **Problem description:** providing coin change of a given amount in the fewest number of coins
- **Inputs:** the dollar-amount to return. Perhaps the possible set of coins, if it is non-obvious.
- **Output:** a set of coins that obtains the desired amount of change in the fewest number of coins
- **Assumptions:** If the coins are not stated, then they are the standard quarter, dime, nickel, and penny. All inputs are non-negative, and dollar amounts are ignored.
- **Strategy:** a greedy algorithm that uses the largest coins first
- **Description:** Issue the largest coin (quarters) until the amount left is less than the amount of a quarter ($0.25). Repeat with decreasing coin sizes (dimes, nickels, pennies).

Another Change Algorithm

- Give me another way to do this?

- Brute force:
  - Generate all possible combinations of coins that add up to the required amount
  - From these, choose the one with smallest number
- What would you say about this approach?

- There are other ways to solve this problem
  - *Dynamic programming*: build a table of solutions to small subproblems, work your way up

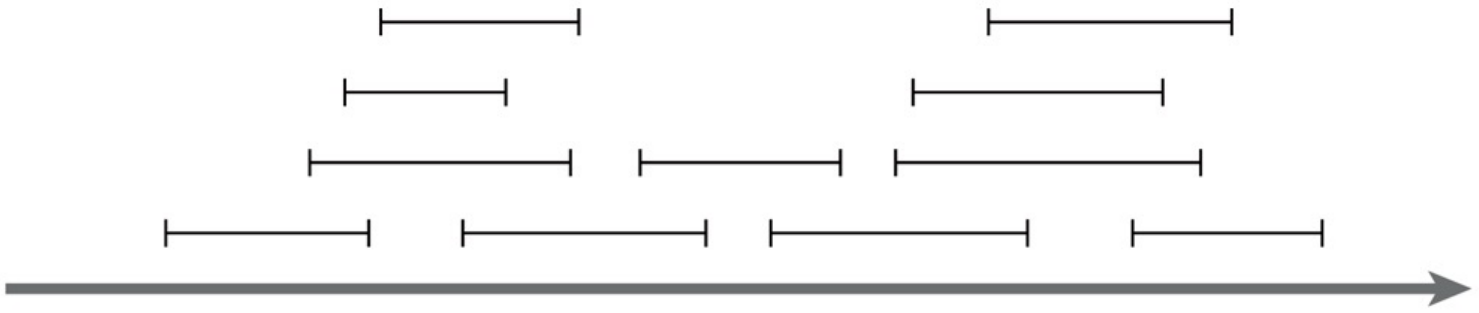
Change solution (brute-force)

- **Problem description:** providing coin change of a given amount in the fewest number of coins
- **Inputs:** the dollar-amount to return. Perhaps the possible set of coins, if it is non-obvious.
- **Output:** a set of coins that obtains the desired amount of change in the fewest number of coins
- **Assumptions:** If the coins are not stated, then they are the standard quarter, dime, nickel, and penny. All inputs are non-negative, and dollar amounts are ignored.
- **Strategy:** a brute-force algorithm that considers every possibility and picks the one with the fewest number of coins
- **Description:** Consider every possible combination of coins that add to the given amount (done via a depth-first search). Return the one with the fewest number of coins.

Motivating problems

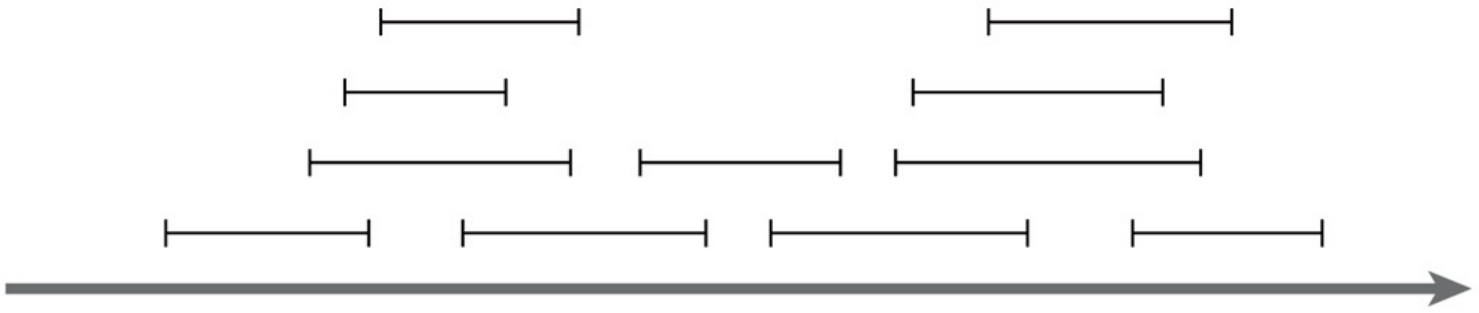
Motivating problem: Interval scheduling

- Interval scheduling
  - Given a series of requests, each with a start time and end time, maximize the number of requests scheduled
  - This is solved by a *greedy* algorithm
  - Most of the CS 2150 algorithms you've seen are greedy algorithms: Dijkstra's shortest path, both MST algorithms, etc.



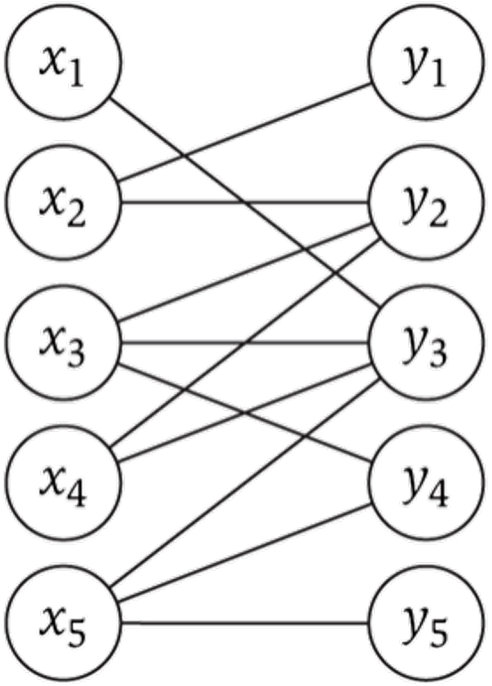
Motivating problem: Weighted interval scheduling

- Weighted interval scheduling
  - Same as the regular interval scheduling, but in addition each request has a cost associated with it
  - The goal is to maximize the cost from scheduling the items
  - This is solved by *dynamic programming*



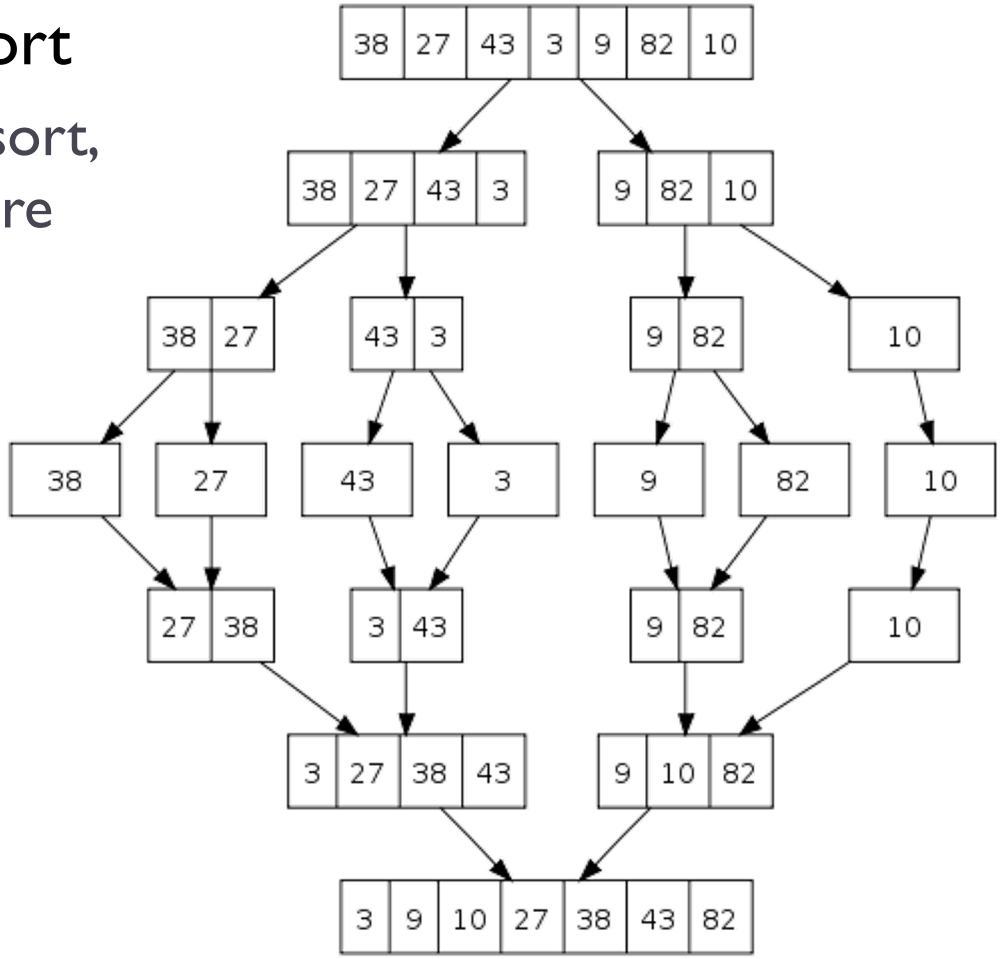
Motivating problem: Bipartite matching

- Bipartite matching
  - Given a graph *G*, find the maximum sub-graph of *G* that partitions *G* into sets *X* and *Y* such that no node from *X* is connected to a node in *Y*, and vise-versa
  - Example: given a series of requests, and entities that can handle each request (such people, computers, etc.), find the optimal matching of requests to entities
  - This is a *network flow* problem



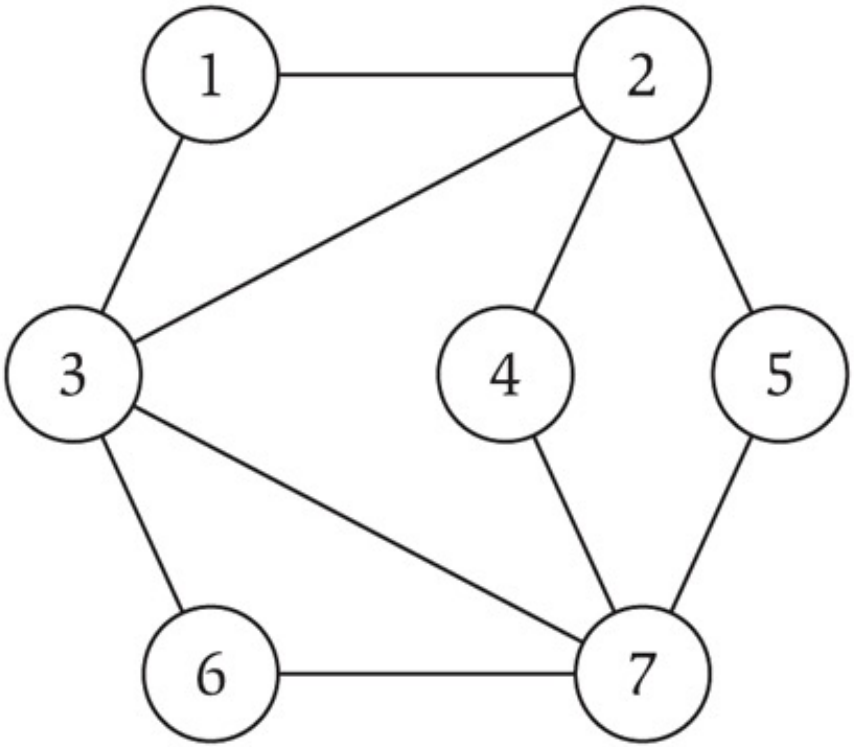
Motivating problem: Sorting

- How do you implement a general-purpose sort that is as efficient as possible in both space and time, and is *stable*?
- One solution is merge-sort
  - We'll see later why quicksort, heapsort, and radix sort are not sufficient
- This is an application of both *sorting* and *divide and conquer*



Motivating problem: Independent set

- Independent set
  - Given a graph *G*, find the maximum size subset *X* of *G* such that no two nodes in *X* are connected to each other
  - This is a *NP-complete* problem
  - You've seen TSP (travelling salesperson problem) in CS 2150, which is a NP-complete problem



Motivating problem: Competitive facility location

- Competitive facility location
  - Consider a graph G, where two 'players' choose nodes in alternating order. No two nodes can be chosen (by either side) if a connecting node is already chosen. Choose the winning strategy for your player.
  - This is a *PSPACE problem*, which are harder than NP-complete problems