WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.
Course goals

In this course you will learn how to:

- translate real-life problems into problems understandable by computers,
- design algorithms to solve problems,
- compare algorithms based on running time,
- write code for algorithms using Python,
- determine when problems are unlikely to be solved by algorithms with reasonable running times, and
- figure out how to come up with “good enough” solutions to such problems.

Note:

- CS 231 complements CS 234.
- You can take CS 231 before or after CS 234.
- CS 234 gives another way to solve problems, using clever data structures (ways of organizing data).
Logistics

Course components and assessment

- Lectures
- Optional Python and math sessions
- Self-checks (in class only)
- 30% Four assignments, mixture of written and programming questions
- 25% Midterm
- 45% Final exam

Resources

- Course material: lecture slides (without verbal and on board explanations) and written resources on selected topics (check dates for latest version)
- The recommended text is optional; use it as a source for more examples and a different presentation of the material.
- Where there are discrepancies, the course material has precedence.
Sources of information and help

General information and help
- Course website
- Piazza
- Office hours

Programming information and help
- Python guide to material to review
- Website links to translations into Python from general programming concepts
- Optional Python sessions
Case study: Trip planning

You are planning a book tour for your best-selling memoir “How I Survived Algorithmic Problem Solving”. What is the best order in which to visit the cities on your list?
Translating for computers

Characteristics of real-life problems:

- Situations may be very complex, involving factors with varying importance.
- It may be difficult to specify what makes one solution better than another.

Characteristics of a computer:

- Tasks need to be specified clearly and precisely.

Making a problem computer friendly:

- Simplify the problem by throwing out some of the information.
- Simplify the way of assessing a solution.
More examples of real-world issues

Challenging issues
- Sets of points
- Objects in space
- Undefined terms
- Precise numbers
Abstract problems

A **problem** is a way of associating an input and an output. It will include:

- an **input specification**, explaining the types of data that make up the parts of the input, and
- an **output specification**, explaining how the output is related to the input.

An **instance** of a problem is obtained by specifying particular values for the parts of the input.
Graphs

A graph consists of vertices connected by edges.

For a graph $G$, the notation $V(G)$ is used for the set of vertices of $G$ and the notation $E(G)$ is used for the set of edges of $G$. We will use $n$ for the size of $V(G)$ and $m$ for the size of $E(G)$.

The two vertices connected by an edge are its endpoints.

We can assign labels, colours, and/or weights to vertices and also to edges.
Trip planning as a graph problem

### Finding a Tour

**Input:** A graph with all possible edges, each with a weight

**Output:** A cycle that includes all the vertices (or a tour)

A cycle is a sequence of vertices such that there is an edge between each consecutive pair of vertices in the sequence and between the last and the first vertex in the sequence.

The total weight of a cycle is the sum of the weights of the edges in the cycle.
Wish list for a graph data type

Useful terms:

- A vertex $u$ is a **neighbour** of a vertex $v$ if there is an edge with endpoints $u$ and $v$.
- If $u$ and $v$ are **neighbours**, they are **adjacent**.
- The number of neighbours of a vertex is the **degree** of the vertex.
- If two edges share an endpoint, the edges are **incident**.
- If a vertex is an endpoint of an edge, the edge and vertex are (also) **incident**.
Module graphs.py

Created for this course; see the resource on graphs for more details
Implementation uses Vertex and Edge objects, accessed by the unique ID of a vertex or the IDs of the two endpoints of the edge.

Example methods and functions:

- `graph.neighbours(one)` produces a list of the IDs of neighbours of the vertex with ID one
- `graph.are_adjacent(one, two)` produces True if vertices with IDs one and two are adjacent and False otherwise
- `graph.add_vertex(one)` and `graph.del_vertex(one)` add and delete vertices. Deleting a vertex automatically deletes all incident edges. There are similar methods for edges.
- `graph.vertex_weight(one)` and `graph.set_vertex_weight(one, weight)` produce and set the weight of vertex one. There are similar methods for labels and colours of vertices, and for weights, labels, and colours of edges.
from graphs import *

def colour_neighbours(graph, target):
    neighbours = graph.neighbours(target)
    for neighbour in neighbours:
        graph.set_vertex_colour(neighbour, "blue")

from graphs import *

def colour_neighbours(graph, target):
    vertices = graph.vertices()
    for vertex in vertices:
        if graph.are_adjacent(vertex, target):
            graph.set_vertex_colour(vertex, "blue")
Types of problems: decision and search

The output of a **decision problem** is the answer (yes or no) to a yes/no question. Each instance is either a **yes-instance** or a **no-instance**.

**Tour Decision**

**Input:** A graph with all possible edges, each with a weight, and a maximum budget $B$

**Output:** Yes or no, answering “Is there a tour of weight at most $B$?”

The output of a **search problem** is an entity that satisfies the specified conditions.

**Tour Search**

**Input:** A graph with all possible edges, each with a weight, and a maximum budget $B$

**Output:** A tour of weight at most $B$
Types of problems: counting and enumeration

The output of a **counting problem** is the number of entities that satisfy the specified conditions.

**Tour Counting**

**Input:** A graph with all possible edges, each with a weight, and a maximum budget B

**Output:** The number of tours of weight at most B

The output of an **enumeration problem** is all the entities that satisfy the specified conditions.

**Tour Enumeration**

**Input:** A graph with all possible edges, each with a weight, and a maximum budget B

**Output:** The set of all tours of weight at most B
An **optimization problem** specifies constraints that are required of any **feasible solution**, a **measure function** that gives a numerical value to each feasible solution, and a **goal** (either maximizing or minimizing the value).

An **optimal solution** has the best value (maximum or minimum, depending on the goal).

Types of problems:

- A **constructive problem** asks for an optimal solution.
- An **evaluation problem** asks for the value of an optimal solution.
Examples of optimization problems

The measure function determines the weight of a tour and the goal is minimization.

**Tour Constructive Optimization**

- **Input:** A graph with all possible edges, each with a weight
- **Output:** A tour of minimum weight

**Tour Evaluation Optimization**

- **Input:** A graph with all possible edges, each with a weight
- **Output:** The minimum weight of a tour
Relations among types of problems

Finding a flight:

- **Decision**  Answers “Is there a flight that costs at most $100?”
- **Search**  Produces a flight that costs at most $100.
- **Counting**  Produces the number of flights that cost at most $100.
- **Enumeration**  Produces all the flights that cost at most $100.
- **Constructive optimization**  Produces the cheapest flight.
- **Evaluation optimization**  Produces the cost of the cheapest flight.
Finding distinct items

Input: A collection of numbers
Output: The numbers in the collection that appear exactly once

Break a problem into smaller problems

- Sort the numbers in the collection
- Scan through the sorted numbers to eliminate duplicates

Use what you already know to sort the numbers, using an algorithm learned in first-year CS.
Paradigm: Exhaustive search

An **exhaustive search** algorithm solves a problem by searching through all possibilities to find a solution.

Recipe for exhaustive search:

1. Determine what the possibilities are.
2. Use the possibilities to solve the problem.

Example: Finding the minimum of a set of numbers.

*What are the possibilities?*

All numbers in the set

*How do we find the solution?*

- Try each number.
- Determine if the number is the minimum comparing it to all the other numbers.
- If it is smaller than all the others, it is the minimum.
Exhaustive search for optimization

**Travelling Salesperson Problem (TSP)**

<table>
<thead>
<tr>
<th>Input:</th>
<th>A graph with all possible edges, each with a weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>A tour of minimum weight</td>
</tr>
</tbody>
</table>

*What are the possibilities?*
All tours

*How do we find the solution?*

**Break a problem into smaller problems:**

- Find all orderings of vertices.
- Find the cost of a single tour.
- Find the minimum of all the costs.

Use what you already know to solve the last problem (either exhaustive search or the better algorithm we used in first-year CS).
You suspect that the creative work that was submitted contains an exact copy of someone else’s work. Is it possible to find one inside the other?

**Gullible variant:** You think that containing a single exact copy could have happened by accident, so you’re really interested in seeing how many copies there are.

**Vigilant variant:** You are not going to be outwitted by a perpetrator who made a few tiny changes to the original work, so you’re going to look for a copy with a limited number of errors in copying.
Grids

Module grids.py supports grids for images and other information.

Example methods and functions:

- Grid(r,c) produces a new empty $r \times c$ grid
- grid.access(row, col) produces the item stored in the given row and column
- grid.enter(row, col, item) enters the item in the given row and column
- make_grid(file) creates a new grid with data from a file
Exhaustive search for a decision problem

**Pattern matching in a grid**

- **Input:** A **pattern** grid and a **target** grid
- **Output:** Yes or no, answering “Does the pattern appear in the target?”

*What are the possibilities?*

All subgrids of the target grid that are the same size as the pattern grid

*How do we find the solution?*

- Try each subgrid.
- Determine if the subgrid matches the pattern grid.
- If any one matches the answer is “yes”; otherwise, the answer is “no”.

---

**CS 231 Module 1**

Case study: Detecting plagiarism 24 / 31
Possibilities for exhaustive search

Minimum: Possibilities were both parts of the input and possible outputs

TSP: Possibilities were both feasible solutions and possible outputs

Pattern matching in a grid: Possibilities were not outputs, but instead can be viewed as witnesses to an output.
Using possibilities to find the solution

**Decision problem**  Check each possibility to see if it is a witness to the answer “yes” or “no” (depending on the problem).

**Search problem**  Check each possibility to see if it satisfies the conditions.

**Counting problem**  Check each possibility to see if it satisfies the conditions, keeping a count of how many qualify.

**Enumeration problem**  Check each possibility to see if it satisfies the conditions, forming output from all that qualify.

**Optimization - constructive**  Determine the value of each feasible solution, returning an optimal solution.

**Optimization - evaluation**  Determine the value of each feasible solution, returning the best value.
Completing the TSP algorithm

Questions to answer:
- How do we determine the cost of a single tour? (See Python session)
- Is the algorithm correct? (Easy to see)
- Is the algorithm fast? (See Module 2)
Topics to follow

- How to choose among algorithms before coding them (Module 2)
- More algorithm paradigms (Modules 3, 4, 5)
- How to determine when a problem is intractable (Module 6)
- Methods to find “good enough” solutions (Modules 7 and 8)
Outline of modules

1. Introduction
2. Comparing problems and solutions
3. Greedy approach
4. Divide-and-conquer
5. Dynamic programming
6. Hardness of problems
7. Compromising on speed
8. Compromising on correctness
Types of problems to follow

- Problems in many different areas
- Problems with many different goals
- Problems on many different kinds of data
Module summary

Topics covered:

- Course goals and logistics
- Case study: Trip planning
- Defining abstract problems
- Encoding an instance
- Graphs, included the module graphs.py
- Types of problems
- Problem-solving approaches
- Paradigm: Exhaustive search
- Case study: Detecting plagiarism
- Grids, including the module grids.py
- Summarizing exhaustive search
- Topics to follow
- Outline of modules