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.
Solving TSP more quickly

**Traveling Salesperson Problem (TSP)**

- **Input:** A graph with all possible edges, each with a weight
- **Output:** A tour of minimum weight
A greedy algorithm usually consists of (optional) preprocessing and then a sequence of steps used to add to the solution. It is usually easy to write.

**Greedy**(*I*)

**INPUT:** An input *I*

**OUTPUT:** A feasible solution to *I*

1. $D \leftarrow$ data from *I*, possibly sorted
2. $S \leftarrow$ partial solution
3. **while** $D$ is not empty
4. $x \leftarrow$ item from $D$
5. remove $x$ and possibly other related items from $D$
6. add $x$ and possibly other related items to $S$
7. **return** $S$
Greedy algorithm advantage: low cost

The cost of a greedy algorithm is typically $O(A + BC)$, where:

- $A$ is the cost of preprocessing the input,
- $B$ is the number of steps, and
- $C$ is the cost of taking a step.

Typically, each of these costs is small with respect to the size of the input.

For TSP:

- There is no preprocessing.
- Each step adds a vertex; $B \in O(n)$.
- Taking a step entails checking the weights of the $O(n)$ edges between the current vertex and the unvisited vertices; $C \in O(n^2)$ since degree of vertex is at most $n$.

The total cost in $O(n^3)$ (actually better is possible), which is much better than $O(n^n)$ for a graph with more than two vertices.
Greedy algorithm disadvantage

Starting at vertex a, we end up with the order a, c, b, d, a for a total cost of $5 + 20 + 6 + 40 = 71$. Is this the best?

Starting at vertex b, we end up with the order b, d, c, a, b for a total cost of $6 + 30 + 5 + 10 = 51$.

Even if we try all the possible starting vertices, the algorithm is not guaranteed to find the best solution.
Case study: Scheduling activities

You have collected information about all the activities that are possible tomorrow, including when each one starts and each one ends. You’ve decided that you’re not willing to arrive late or leave early, but still want to fit in as many activities as possible. What schedule should you choose?

Scheduling activities

Input: A set of $n$ activities, where activity $i$ has start time $s_i$ and finish time $f_i$

Output: A schedule that has the maximum number of nonoverlapping activities, that is, for each pair chosen $f_i \leq s_j$ or $f_j \leq s_i$

Aside

There are many variants of this problem: maximize the time spent on activities, is in use, allow parts of activities to be scheduled, and so on.
Choose shortest activities first

Idea: If you choose short activities, you can fit in more of them.

Choose the shortest activity, then the next shortest non-overlapping activity, and so on.
Choose activities by earliest finishing time

Idea: If you choose activities that finish soon, you’ll have more time to fit in others.

Choose the activity that finishes first, then the non-overlapping activity that finishes next earliest, and so on.
Greedy algorithm analysis

Algorithm analysis:
- Determine the worst-case running time (usually easy).
- Show that the algorithm produces a feasible solution.
- Show that the algorithm produces an optimal solution.

In lectures and on assignments we will not provide formal proofs.
“Finish first” design and running time analysis

1. \[ A \leftarrow \text{activities sorted by nondecreasing finishing time} \]
2. \[ S \leftarrow \emptyset \]
3. while \( A \) is not empty
4. \[ x \leftarrow \text{any activity in } A \text{ with smallest finishing time} \]
5. \[ S \leftarrow S \cup \{x\} \]
6. \[ A \leftarrow A \setminus \{\text{activities that overlap } x\} \]

Worst-case running time:

Step 1: \( O(n \log n) \) time (to be discussed more later)

Step 2: \( O(1) \) time

Steps 3-6: Process \( A \) in order, \( O(n) \) items in total, \( O(n) \) to process each item (actually faster is possible)

Total time: \( O(n^2) \)
“Finish first” produces a feasible solution

Each time we select an activity, we remove all activities that conflict with it.

None of the selected activities conflict with each other.

The solution is feasible (there are no conflicts).
Showing optimality

“Greedy stays ahead” argument
Show that the greedy solution stays ahead of the optimal solution

- Find measures by which the algorithm’s solution is at least as good as any other solution.
- Show that the measures imply the algorithm’s solution is optimal.

“Exchange” argument
Show that you can iteratively transform an optimal solution to the one formed by the greedy algorithm.

- Find a difference between your solution $A$ and the optimal solution $O$, such as a different element or a different order.
- Make a change, such as of elements or orderings, to eliminate the difference.
- Show that the change does not make the solution $A$ any worse than it was.
“Finish first” produces an optimal solution

$A = \{a_1, \ldots, a_j\}$ is the solution formed by our algorithm
$O = \{o_1, \ldots, o_k\}$ is an optimal solution

Measure: For each value $1 \leq i \leq j$, the finish time of item $i$.

To show:

- Argue that $f(a_i) \leq f(o_i)$ for all values $1 \leq i \leq j$.
- Argue that as a consequence, $A$ must be optimal.

Sketch of arguments:

- This follows from the way items are selected by the algorithm.
- If $A$ is not optimal, then another item could have been added to the schedule. However, $O$’s last item finished no sooner than $A$’s last item.
Implementing the algorithm

- Represent activities
- Sort activities by finish time
- Check for overlaps
- Form the schedule
Representing activities

class Activity:
    def __init__(self, name, start, end):
        self.name = name
        self.start = start
        self.end = end

(Code available from course website as module3activity.py with testing file module3activityuse.py.)
Sorting activities by finish time

Sorting a Python list:

- `a_list.sort()` mutates `a_list`
- `sorted(a_list)` produces a new list
- `reverse` can be used to sort in nonincreasing order
- `key` can be used to specify how to sort

Example:
`sorted(a_list, key=my_func, reverse=True)`
Checking for overlaps

The following method is part of the class definition for Activity:

```python
def overlap(self, other):
    if self.start <= other.start and
        self.end > other.start:
        return True
    elif other.start <= self.start and
        other.end > self.start:
        return True
    else:
        return False
```
Forming the schedule

Use list operations to form the schedule.

Try writing the entire algorithm as an exercise.

(Code available on the course website as module3finishfirst.py, including an optional modification using filter and lambda.)
Making change

**Making change**

*Input:* A set of integer coin values \(c_1, c_2, \ldots, c_k\) and an integer \(x\)

*Output:* Numbers of coins of each denomination such that the total number of coins is the smallest possible and the total value is \(x\)

Example: Make change of $3.96 using coins with denominations 1, 5, 10, and 25 cents, one dollar, and two dollars.
Greedy algorithm for making change

Recipe for a greedy algorithm:

• Build up the solution in steps. At each step, add one coin to the collection.

• Choose the next part of the solution in a “greedy” manner. Choose the maximum denomination coin less than or equal to the remaining change to be made.

• Once a decision has been made, never undo it. Do not remove a coin once chosen.

Is this correct for the example?
Is this correct for any choice of denominations?
Greedy sorting

**Sorting**

Input: A set of numbers
Output: The numbers in order from smallest to largest

Recipe for a greedy algorithm:

- Build up the solution in steps. *At each step, add an item to a sequence of items in sorted order.*
- Choose the next part of the solution in a “greedy” manner. *Choose the minimum item among the unordered items. Use what you already know* for this.
- Once a decision has been made, never undo it. *Do not change the order of previously-placed items.*

Sorting in place:

- In phase $i$, put the minimum element in positions $i$ through $n$ into position $i$.
- At the end of phase $i$, items in the first $i$ positions are in order.
Selection sort

INPUT

4 2 6 3 1 5

```
def select_sort(alist):
    length = len(alist)
    for phase in range(length):
        smallest_so_far = phase
        for choice in range(phase+1, length):
            if alist[choice] < alist[smallest_so_far]:
                smallest_so_far = choice
        temp = alist[phase]
        alist[phase] = alist[smallest_so_far]
        alist[smallest_so_far] = temp
    return alist
```
Selection sort

PHASE 1

| 1 | 2 | 6 | 3 | 4 | 5 |

1 def select_sort(alist):
2    length = len(alist)
3    for phase in range(length):
4        smallest_so_far = phase
5        for choice in range(phase+1, length):
6            if alist[choice] < alist[smallest_so_far]:
7                smallest_so_far = choice
8        temp = alist[phase]
9        alist[phase] = alist[smallest_so_far]
10       alist[smallest_so_far] = temp
11    return alist
def select_sort(alist):
    length = len(alist)
    for phase in range(length):
        smallest_so_far = phase
        for choice in range(phase+1, length):
            if alist[choice] < alist[smallest_so_far]:
                smallest_so_far = choice
        temp = alist[phase]
        alist[phase] = alist[smallest_so_far]
        alist[smallest_so_far] = temp
    return alist
Selection sort

1  def select_sort(alist):
2      length = len(alist)
3      for phase in range(length):
4          smallest_so_far = phase
5          for choice in range(phase+1, length):
6              if alist[choice] < alist[smallest_so_far]:
7                  smallest_so_far = choice
8          temp = alist[phase]
9          alist[phase] = alist[smallest_so_far]
10         alist[smallest_so_far] = temp
11     return alist
Selection sort

PHASE 4

1 2 3 4 6 5

1 def select_sort(alist):
2     length = len(alist)
3     for phase in range(length):
4         smallest_so_far = phase
5         for choice in range(phase+1, length):
6             if alist[choice] < alist[smallest_so_far]:
7                 smallest_so_far = choice
8         temp = alist[phase]
9         alist[phase] = alist[smallest_so_far]
10        alist[smallest_so_far] = temp
11     return alist
Selection sort

PHASE 5

1 2 3 4 5 6

1 def select_sort(alist):
2     length = len(alist)
3     for phase in range(length):
4         smallest_so_far = phase
5         for choice in range(phase+1, length):
6             if alist[choice] < alist[smallest_so_far]:
7                 smallest_so_far = choice
8         temp = alist[phase]
9         alist[phase] = alist[smallest_so_far]
10        alist[smallest_so_far] = temp
11     return alist
You would like to bring various supplies to your family overseas, but you are constrained by the weight limit for your luggage. What should you pack and what should you leave behind?

Aside

Later in the course we will consider other variants of this problem.
Fractional knapsack

**Fractional knapsack**

**Input:** A set of \( n \) types of objects, where object \( i \) has integer weight \( w_i \) and integer value \( v_i \), and an integer weight bound \( W \)

**Output:** A list of fractions \( 0 \leq x_i \leq 1 \) such that \( \sum_{i=1}^{n} x_i \cdot w_i \leq W \) and \( \sum_{i=1}^{n} x_i \cdot v_i \) is maximized

---

**Example:** \( n = 3, \ W = 70 \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( w_i )</th>
<th>( v_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>4</td>
</tr>
</tbody>
</table>

value \( 100 \times 1 = 100 \)

value \( 30 \times 2/3 = 20 \)
Greedy algorithm for knapsack

Recipe for a greedy algorithm:

• Build up the solution in steps. *At each step, add a fraction of an item.*
• Choose the next part of the solution in a “greedy” manner. *Order the items in some way*
• Once a decision has been made, never undo it. *Do not remove an item (or a piece of an item) once it has been placed in the knapsack.*
Choosing highest value first

Order the objects from highest value to lowest
Take as much as possible of the objects in order

Example: \( n = 5, \ W = 100 \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( w_i )</th>
<th>( v_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>66</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

\[ \text{66} \quad \text{60} \quad \text{40} \times 1/2 \]
Choosing lowest weight first

Order the objects from lowest weight to highest
Take as much as possible of the objects in order

Example: $n = 5$, $W = 100$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$w_i$</th>
<th>$v_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>66</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

20 30 66 40
Greedy solutions compared

Choosing highest value first gives a total of 146:

Choosing lowest weight first gives a total of 156:

The best solution is a total of 164:
Choosing best ratio of value to weight

Idea: Get as much value as possible for each unit of weight.
Order the objects by nonincreasing ratio of value to weight
Take as much as possible of the objects in order

Running time:
Cost of sorting: $O(n \log n)$
Cost of trying objects: $O(n)$
Optimal substructure

A problem has **optimal substructure** if the solution to the original instance contains within it the optimal solution to a smaller instance.

Examples we’ve seen so far:

- Scheduling activities
- Making change
- Sorting
- Knapsack
As a frequent traveller, you wish to save money by determining the cheapest way of getting to each of the destinations you visit.

**Input:** A graph $G$ with non-negative edge weights and a source vertex $s \in V(G)$

**Output:** Least-cost paths from $s$ to each vertex in $G$, where the cost is the sum of the weights of edges in the path
Dijkstra’s algorithm

Maintain:

• $K$, the set of vertices such that cheapest paths from $s$ are known
• $U$, the set of vertices not in $K$
• $\text{dist}(v)$ for all $v \in U$, the cheapest cost of a special path from $s$ to $v$ using only vertices in $K$ as internal nodes on the path

Algorithm idea:

• Repeatedly (until $U$ is empty), choose a $v$ in $U$ with the cheapest special path.
• Add $v$ to $K$
• Update values of dist for all remaining vertices in $U$
Example of Dijkstra
Example of Dijkstra
Example of Dijkstra
Example of Dijkstra

Diagram of a network with nodes labeled a, b, c, d, e, f, and g, and edges with weights labeled 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 15, and 18.
Example of Dijkstra
Example of Dijkstra

\[\begin{array}{cccccccc}
a & 15 & 12 & 6 & 7 & 11 & 4 & 9 & 3 \\
b & 5 & 11 & 1 & 7 & 7 & 18 & 7 & 11 \\
c & 15 & 12 & 6 & 6 & 3 & 11 & 7 & 9 \\
d & 5 & 11 & 1 & 6 & 6 & 11 & 7 & 9 \\
e & 7 & 7 & 3 & 3 & 9 & 4 & 7 & 7 \\
f & 11 & 7 & 11 & 7 & 7 & 4 & 3 & 3 \\
g & 15 & 11 & 7 & 7 & 11 & 11 & 7 & 9 \\
\end{array}\]
Example of Dijkstra
You wish to set up a communications network that allows you to reach each entity from each other entity. There are costs associated with links between pairs of entities. How can you find a cheapest way of forming a network?
Minimum spanning tree

Definitions

A *spanning tree* is a tree connecting all vertices in a connected graph, formed of a subset of the edges in the graph. A *minimum spanning tree* is a spanning tree such that the sum of the weights of the edges is minimized.

**Minimum spanning tree**

**Input:** A connected graph $G$ with non-negative edge weights

**Output:** A subset $A$ of $E(G)$ that forms a tree from all of $V(G)$ and the sum of the weights of edges in $A$ is as small as possible
Greedy approach to minimum spanning tree

Recipe for a greedy algorithm:

- Build up the solution in steps. *At each step, add an edge to edges that can be formed into a tree.*
- Choose the next part of the solution in a “greedy” manner. *Choose an edge that is cheapest according to some criterion.*
- Once a decision has been made, never undo it. *Do not remove an edge once it has been selected.*

1. \( A \leftarrow \emptyset \)
2. while \( A \) does not form a spanning tree
3. find an edge \( e \) of least cost with certain properties
4. \( A \leftarrow A \cup \{e\} \)

We will consider various options for line 3.
Kruskal’s Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.
Kruskal’s Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.
Kruskal’s Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.
Kruskal’s Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.
Kruskal’s Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.
Kruskal’s Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.
Kruskal’s Algorithm

- Start with the empty set of edges
- Add the lowest weight edge that does not form a cycle.
Prim’s Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.
Prim’s Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.
Prim’s Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.
Prim’s Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.
Prim’s Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.
Prim’s Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.
Prim’s Algorithm

- Start with a single vertex.
- Add the lowest weight edge that joins a vertex outside of the tree formed by A to the tree formed by A.
Analysis of minimum spanning tree algorithms

Sophisticated ways of manipulating data are used keep the running time low.

Details are outside the scope of this course.

Kruskal’s algorithm: $O(m \log m)$ time
Prim’s algorithm: $O(m \log n)$ time
Case study: Colouring a map

You wish to colour your historical map of the world in such a way that no two adjacent countries receive the same colour.

A map can be expressed as a graph.

To colour a map, each vertex represents a country, and an edge indicates that the countries corresponding to its endpoints are adjacent. The endpoints of an edge must be assigned different colours.

We can also look at colouring of graphs that do not represent maps.
Colouring

A graph is *k-colourable* if each vertex can be assigned one of *k* colours such that each edge has endpoints of two different colours.

**k-Colouring**

*Input:* A graph *G* and an integer *k*

*Output:* Yes or no, answering “Is *G* is *k*-colourable?”
Greedy 3-colouring

**3-Colouring**

**Input:** A graph $G$

**Output:** Yes or no, answering “Is $G$ is 3-colourable?”

Notice that the number three is not part of the input.

We may consider greedy approaches to this problem in assignments and/or exams.
Comparing paradigms

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Exhaustive search</th>
<th>Greedy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subproblems</td>
<td>All</td>
<td>Some</td>
</tr>
<tr>
<td>Applicability</td>
<td>Wide</td>
<td>Narrow</td>
</tr>
<tr>
<td>Speed</td>
<td>Slow</td>
<td>Fast</td>
</tr>
</tbody>
</table>

Properties of greedy algorithms:

- Each step of the algorithm eliminates some possible solutions from consideration, as no decision is ever “undone”
- Running time analysis is often simple
- Proving correctness can be difficult (use “greedy stays ahead” and “exchange” arguments)
- Proving an algorithm is not correct can be achieved with a single counterexample
Module summary

Topics covered:

- Paradigm: Greedy algorithms
- Greedy TSP
- Case study: Scheduling activities
- Making change
- Greedy sorting
- Fractional knapsack
- Case study: Colouring a map
- Greedy colouring
- Single-source cheapest paths
- Case study: Designing a network
- Greedy minimum spanning tree
- Comparing paradigms