Greedy Algorithms
Introduction

Similar to dynamic programming.
Used for optimization problems.

Idea: When we have a choice to make, make the one that looks best right now. Make a locally optimal choice in hope of getting a globally optimal solution.
Greedy algorithms don’t always yield an optimal solution. But sometimes they do. We’ll see a problem for which they do. Then we’ll look at some general characteristics of when greedy algorithms give optimal solutions.
Activity Selection

\( n \) activities require exclusive use of a common resource. For example, scheduling the use of a classroom.

Set of activities \( S = \{a_1, \ldots, a_n\} \).

\( a_i \) needs resource during period \([s_i, f_i]\), which is a half-open interval, where \( s_i = \) start time and \( f_i = \) finish time.

**Goal:** Select the largest possible set of nonoverlapping (mutually compatible) activities.

**Note:** Could have many other objectives:

- Schedule room for longest time.
- Maximize income rental fees.
Example: \( S \) sorted by finish time:

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_i )</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>( f_i )</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Maximum-size mutually compatible set: \( \{a_1, a_3, a_6, a_8\} \).
Not unique: also \( \{a_2, a_5, a_7, a_9\} \).
Optimal Substructure of Activity Selection

\[ S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\} \]

= activities that start after \( a_i \) finishes and finish before \( a_j \) starts.

\[ \cdots f_i \quad s_k \quad f_k \quad s_j \quad \cdots \]
\[ a_i \quad a_k \quad a_j \quad \cdots \]

Activities in \( S_{ij} \) are compatible with

- all activities that finish by \( f_i \), and
- all activities that start no earlier than \( s_j \).
To represent the entire problem, add fictitious activities:

\[ a_0 = [-\infty, 0) \]
\[ a_{n+1} = [\infty, \infty + 1] \]

We don’t care about \(-\infty\) in \(a_0\) or \(\infty + 1\) in \(a_{n+1}\).

Then \(S = S_{0,n+1}\).

Range for \(S_{ij}\) is \(0 \leq i, j \leq n + 1\).

Assume that activities are sorted by monotonically increasing finish time:

\[ f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1} . \]

Then \(i \geq j \Rightarrow S_{ij} = \emptyset\).

- If there exists \(a_k \in S_{ij}\):
  \[ f_i \leq s_k < f_k \leq s_j < f_j \Rightarrow f_i < f_j . \]
- But \(i \geq j \Rightarrow f_i \geq f_j\). Contradiction.
So only need to worry about $S_{ij}$ with $0 \leq i < j \leq n + 1$. All other $S_{ij}$ are $\emptyset$.

Suppose that a solution to $S_{ij}$ includes $a_k$. Have 2 subproblems:
- $S_{ik}$ (start after $a_i$ finishes, finish before $a_k$ starts)
- $S_{kj}$ (start after $a_k$ finishes, finish before $a_j$ starts)

Solution to $S_{ij}$ is $(\text{solution to } S_{ik}) \cup \{a_k\} \cup (\text{solution to } S_{kj})$.

Since $a_k$ is in neither subproblem, and the subproblems are disjoint,
$|\text{solution to } S| = |\text{solution to } S_{ik}| + 1 + |\text{solution to } S_{kj}|$.

If an optimal solution to $S_{ij}$ includes $a_k$, then the solutions to $S_{ik}$ and $S_{kj}$ used within this solution must be optimal as well. Use the usual cut-and-paste argument. Let $A_{ij} = \text{optimal solution to } S_{ij}$.

So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, assuming:
- $S_{ij}$ is nonempty, and
- we know $a_k$. 
Recursive solution

c[i, j] = size of maximum-size subset of mutually compatible activities in S_{ij}.

- i ≥ j ⇒ S_{ij} = ∅ ⇒ c[i, j] = 0.

If S_{ij} ≠ ∅, suppose we know that a_k is in the subset. Then

\[ c[i, j] = c[i, k] + 1 + c[k, j]. \]

But of course we don’t know which k to use, and so

\[ c[i, j] = \begin{cases} 
0 & \text{if } S_{ij} = \emptyset, \\
\max_{i<k<j, a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset.
\end{cases} \]

Why this range of k? Because S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\} ⇒ a_k can’t be a_i or a_j. Also need to ensure that a_k is actually in S_{ij}, since i < k < j is not sufficient on its own to ensure this.

From here, we could continue treating this like a dynamic-programming problem. We can simplify our lives, however.
Theorem
Let $S_{ij} \neq \emptyset$, and let $a_m$ be the activity in $S_{ij}$ with the earliest finish time: $f_m = \min \{ f_k : a_k \in S_{ij} \}$. Then:

1. $a_m$ is used in some maximum-size subset of mutually compatible activities of $S_{ij}$.
2. $S_{im} = \emptyset$, so that choosing $a_m$ leaves $S_{mj}$ as the only nonempty subproblem.
Recursive solution

This is great:

- # of subproblems in optimal solution
  - before theorem: 2
  - after theorem: 1
- # of choices to consider
  - before theorem: \( j - i - 1 \)
  - after theorem: 1

Now we can solve *top down*:

- To solve a problem \( S_{ij} \),
  - Choose \( a_m \in S_{ij} \) with earliest finish time: the *greedy choice*.
  - Then solve \( S_{mj} \).
Recursive solution

What are the subproblems?

- Original problem is $S_{0,n+1}$.
- Suppose our first choice is $a_{m_1}$.
- Then next subproblem is $S_{m_1,n+1}$.
- Suppose next choice is $a_{m_2}$.
- Next subproblem is $S_{m_2,n+1}$.
- And so on.

Each subproblem is $S_{m_i,n+1}$, i.e., the last activities to finish.

And the subproblems chosen have finish times that increase.

Therefore, we can consider each activity just once, in monotonically increasing order of finish time.
// Assumes sorted input

\textbf{Rec-Activity-Selector}(s, f, i, n) \\
\hspace{1em} m \leftarrow i + 1 \\
\hspace{1em} \textbf{while } m \leq n \text{ and } s_m < f_i \hspace{1em} \triangleright \text{ Find first activity in } S_{i,n+1}. \\
\hspace{2em} \textbf{do } m \leftarrow m + 1 \\
\hspace{1em} \textbf{if } m \leq n \\
\hspace{2em} \textbf{then return } \{a_m\} \cup \textbf{Rec-Activity-Selector}(s, f, m, n) \\
\hspace{2em} \textbf{else return } \emptyset

\textbf{Initial call: } \textbf{Rec-Activity-Selector}(s, f, 0, n).

\textbf{Idea: } The \textbf{while} loop checks \(a_{i+1}, a_{i+2}, \ldots, a_n\) until it finds an activity \(a_m\) that is compatible with \(a_i\) (need \(s_m \geq f_i\)).

- If the loop terminates because \(a_m\) is found \((m \leq n)\), then recursively solve \(S_{m,n+1}\), and return this solution, along with \(a_m\).
- If the loop never finds a compatible \(a_m\) \((m > n)\), then just return empty set.

Go through example given earlier. Should get \(\{a_1, a_4, a_8, a_{11}\}\).

\textbf{Time: } \Theta(n)—each activity examined exactly once.
Figure 16.1 The operation of \( \text{RECURSIVE-ACTIVITY-SELECTOR} \) on the 11 activities given earlier. Activities considered in each recursive call appear between horizontal lines. The fictitious activity \( f_0 \) finishes at time 0, and in the initial call, \( \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, 0, 12) \), activity \( a_1 \) is selected. In each recursive call, the activities that have already been selected are shaded, and the activity shown in white is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, \( \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, 11, 12) \), returns \( \emptyset \). The resulting set of selected activities is \{\( a_1, a_4, a_8, a_{11} \)\}. 
Can make this iterative. It’s already almost tail recursive.

**GREEDY-ACTIVITY-SELECTOR** \((s, f, n)\)

\(A \leftarrow \{a_1\}\)

\(i \leftarrow 1\)

\textbf{for} \(m \leftarrow 2 \textbf{ to } n\)

\textbf{do if} \(s_m \geq f_i\)

\textbf{then} \(A \leftarrow A \cup \{a_m\}\)

\(i \leftarrow m\) \quad \triangleright a_i \text{ is most recent addition to } A\)

\textbf{return} \(A\)

Go through example given earlier. Should again get \(\{a_1, a_4, a_8, a_{11}\}\).

**Time:** \(\Theta(n)\).
Elements of the greedy strategy

The choice that seems best at the moment is the one we go with.

What did we do for activity selection?

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Prove that at any stage of recursion, one of the optimal choices is the greedy choice. Therefore, it’s always safe to make the greedy choice.
4. Show that all but one of the subproblems resulting from the greedy choice are empty.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.
Elements of the greedy strategy

At first, it looked like dynamic programming. Typically, we streamline these steps. Develop the substructure with an eye toward

- making the greedy choice,
- leaving just one subproblem.

For activity selection, we showed that the greedy choice implied that in $S_{ij}$, only $i$ varied, and $j$ was fixed at $n + 1$.

We could have started out with a greedy algorithm in mind:

- Define $S_i = \{a_k \in S : f_i \leq s_k\}$.
- Then show that the greedy choice—first $a_m$ to finish in $S_i$—combined with optimal solution to $S_m \Rightarrow$ optimal solution to $S_i$. 

Elements of the greedy strategy

Typical streamlined steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there’s always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Show that greedy choice and optimal solution to subproblem ⇒ optimal solution to the problem.

No general way to tell if a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.
Greedy Vs. Dynamic

**Greedy-choice property**

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

**Dynamic programming:**

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems first.
- Solve bottom-up.
Greedy Vs. Dynamic

**Greedy:**

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*.

Typically show the greedy-choice property by what we did for activity selection:

- Look at a globally optimal solution.
- If it includes the greedy choice, done.
- Else, modify it to include the greedy choice, yielding another solution that’s just as good.

Can get efficiency gains from greedy-choice property.

- Preprocess input to put it into greedy order.
- Or, if dynamic data, use a priority queue.
An example

0-1 knapsack problem:

- $n$ items.
- Item $i$ is worth $v_i$, weighs $w_i$ pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- Have to either take an item or not take it—can’t take part of it.

Fractional knapsack problem: Like the 0-1 knapsack problem, but can take fraction of an item.

Both have optimal substructure.

But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.

To solve the fractional problem, rank items by value/weight: $v_i/w_i$.

Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all $i$. 
**Time:** \(O(n \lg n)\) to sort, \(O(n)\) thereafter.

Greedy doesn’t work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.

<table>
<thead>
<tr>
<th>(i)</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_i)</td>
<td>60</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>(w_i)</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>(v_i/w_i)</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

\(W = 50\).

Greedy solution:

- Take items 1 and 2.
- value = 160, weight = 30.

Have 20 pounds of capacity left over.

Optimal solution:

- Take items 2 and 3.
- value = 220, weight = 50.

No leftover capacity.
Figure 16.2  The greedy strategy does not work for the 0-1 knapsack problem.  (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds.  (b) The optimal subset includes items 2 and 3.  Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound.  (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.