Greedy Algorithms
On this lecture

- Definition of a Greedy Algorithm
- Coin Change (cashier problem)
- Properties of a Greedy Algorithm
  - Optimal Substructure
  - Greedy Choice Property
- Fractional Knapsack
- Interval Scheduling
- Minimum Cost Sum
Greedy Algorithm

- At each step choose the "best" local choice
- Never look "behind" or change any decisions already made
- Never look to the "future" to check if our decision negative consequences
Greedy Algorithms
A first example

**Coin Change Problem (Cashier’s Problem)**

**Input:** a set of coins $S$ and a quantity $K$ we want to create with the coins

**Output:** the minimum number of coins to make the quantity $K$ (we can repeat coins)

**Input/Output Example**

**Input:** $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$
(we have an infinite supply of each coin)

$K = 42$

**Output:** 3 coins ($20 + 20 + 2$)
Coin Change Problem

**A greedy algorithm for the coin change problem**

In each step choose the largest coin that we will not take us past quantity \( K \).

Examples (with \( S = \{1, 2, 5, 10, 20, 50, 100, 200\} \)):

- \( K = 35 \)
  - \( 20 \) (total: 20) + \( 10 \) (total: 30) + \( 5 \) (total: 35) [3 coins]

- \( K = 38 \)
  - \( 20 + 10 + 5 + 2 + 1 \) [5 coins]

- \( K = 144 \)
  - \( 100 + 20 + 20 + 2 + 2 \) [5 coins]

- \( K = 211 \)
  - \( 200 + 10 + 1 \) [3 coins]
Coin Change Problem

- Does this algorithm always give the **minimum** amount of coins?
- For the common money systems (ex: euro, dólar)... **yes**!
- For a general coin set... **no**!

Examples:

- \( S = \{1, 2, 5, 10, 20, 25\} \), \( K = 40 \)
  - Greedy gives 3 coins \((25 + 10 + 5)\), but it is possible to use 2 \((20 + 20)\)
- \( S = \{1, 5, 8, 10\} \), \( K = 13 \)
  - Greedy gives 4 coins \((10 + 1 + 1 + 1)\), but it is possible to use 2 \((5 + 8)\)

(Will it be enough that a single coin is larger than the double of the previous coin?)

- \( S = \{1, 10, 25\} \), \( K = 40 \)
  - Greedy gives 7 coins \((25 + 10 + 1 + 1 + 1 + 1 + 1)\), but is is possible to only use 4 coins \((10 + 10 + 10 + 10)\)
"Simple" idea, but it does not always work
- Depending on the problem, it may or may not give an optimal answer

Normally, the **running time is very low** (ex: linear or linearithmic)

The **hard part** is to prove optimality

Typically is it applied in **optimization problems**
- Find the "best" solution among all possible solutions, according to a given criteria (goal function)
- Generally it involves finding a minimum or a maximum

A very common pre-processing step is... **sorting**!
## Optimal Substructure

When the optimal solution of a problem contains in itself solutions for subproblems of the same type.

## Example

Let $min(K)$ be the minimum amount of coins to make quantity $K$. If that solution uses a coin of value $v$, then the remaining coins to use are given precisely by $min(K - v)$.

- If a problem presents this characteristic, we say it respects the **optimality principle**.
Properties needed for a greedy approach to work

Greedy Choice Property
An optimal solution is consistent with the greedy choice of the algorithm.

Example
In the case of euro coins, there is an optimal solution using the largest coin which is still smaller or equal than the quantity we need to make.

• Proving this property is sometimes a bit complicated.
Fractional Knapsack

Fractional Knapsack Problem

**Input:** A backpack of capacity $C$
A set of $n$ materials, each one with weight $w_i$ and value $v_i$

**Output:** The allocation of materials to the backpack that maximizes the transported value.

The materials can be ”broken” em smaller pieces, that is, we can decide to take only quantity $x_i$ of object $i$, with $0 \leq x_i \leq 1$.

What we want is therefore to obey the following constraints

- The materials fit in the backpack ($\sum_i x_i w_i \leq C$)
- The value transported is the maximum possible (maximize $\sum_i x_i v_i$)
Fractional Knapsack

**Input Example**

**Input:** 5 objects e C = 100

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

What is the optimal answer in this case?

- Always choose the material with the largest value:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_i</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.5</td>
<td>1</td>
</tr>
</tbody>
</table>

This would give a total weight of 100 and a total value of 146.
Fractional Knapsack

**Input Example**

**Input:** 5 objects e $C = 100$

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

What is the optimal answer in this case?

- Always choose the material with the smallest weight:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

This would give a total weight of 100 and a total value of 156.
Fractional Knapsack

Input Example

Input: 5 objects e \( C = 100 \)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>( w_i )</td>
<td>( v_i )</td>
<td>( v_i/w_i )</td>
<td>( x_i )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>20</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>30</td>
<td>1.5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>66</td>
<td>2.2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>40</td>
<td>1.0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>60</td>
<td>1.2</td>
<td>0.8</td>
<td></td>
</tr>
</tbody>
</table>

What is the optimal answer in this case?

- Always choose the material with the largest value/weight ratio:

This would give a total weight of 100 and a total value of 164.
Theorem

Always choosing the largest possible quantity of the material with the largest value/weight ratio is strategy leading to an optimal total value.

1) Optimal Substructure

Consider an optimal solution and its material $m$ with the best ratio.

If we remove it from the backpack, then the remaining objects must contain the optimal solution for the materials other than $m$ and for a backpack with capacity $C - w_m$

If that is not the case, then the initial solution was also not optimal!
Fractional Knapsack

**Theorem**

Always choosing the largest possible quantity of the material with best \(\text{value/weigth}\) ratio is a strategy that gives an optimal value.

2) **Greedy Choice Property**

We want to prove that the largest possible quantity of the material \(m\) with the best ratio \((v_i/w_i)\) should be included in the backpack.

The value of the backpack: 

\[
\text{value} = \sum_i x_i v_i.
\]

Let \(q_i = x_i w_i\) be the quantity of material \(i\): 

\[
\text{value} = \sum_i q_i v_i / w_i.
\]

If we still have some material \(m\) available, then swapping any other material \(i\) with \(m\) will give origin to a better total value: 

\[
q_i v_m / w_m \geq q_i v_i / w_i \text{ (by definition of } m)\]

**Fractional Knapsack**

---

**Greedy Algorithm for Fractional Knapsack**

- Sort the materials by decreasing order of \( \text{value/weigth} \) ratio
- Process the next material in the sorted list:
  - If it fits entirely on the backpack, include it all and continue to the next material
  - If it does not fit entirely, include the largest possible quantity and terminate

---

**Temporal Complexity:**

- Sorting: \( O(n \log n) \)
- Processing: \( O(n) \)
- Total: \( O(n \log n) \)
Interval Scheduling Problem

**Input:** A set of $n$ activities, each one starting on time $s_i$ and finishing on time $f_i$.

**Output:** Largest possible quantity of activities without overlapping
Two intervals $i$ and $j$ overlap if there is a time $k$ where both are active.
Interval Scheduling

Input Example

Input: 5 activities:

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$f_i$</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Pedro Ribeiro (DCC/FCUP)
Greedy “pattern”: Establish an order according to a certain criteria and then choose activities that do not overlap with activities already chosen

Some possible ideas:

- [Earliest start] Allocate by increasing order of $s_i$
- [Earliest finish] Allocate by increasing order of $f_i$
- [Smallest interval] Allocate by increasing order of $f_i - s_i$
- [Smallest number of conflicts] Allocate by increasing order of the number of activities that overlap with it
Interval Scheduling

[Earliest start] Allocate by increasing order of $s_i$

Counter-Example:
Interval Scheduling

[Smallest interval] Allocate by increasing order of $f_i - s_i$

Counter-Example:
Interval Scheduling

[Smallest number of conflicts] Allocate by increasing order of the number of activities that overlap with it

Counter-Example:
Interval Scheduling

[Earliest finish] Allocate by increasing order of $f_i$

**Counter-Example:** Does not exist!
In fact, this greedy strategy produces an optimal solution!

**Theorem**
Always choose the non-overlapping activity with the smallest possible finish time will produce an optimal solution

1) Optimal substructure
Consider an optimal solution and the activity $m$ with the smallest $f_m$.
If we remove that activity, then the remaining activities must contain an optimal solution for all activities starting after $f_m$.
If that is not the case, then the initial solution would not be optimal!
Interval Scheduling

**Theorem**
Always choose the non-overlapping activity with the smallest possible finish time will produce an optimal solution.

2) **Greedy Choice Property**
Let’s assume that the activities are sorted by increasing order of finish times.

Let $G = \{g_1, g_2, \ldots, g_m\}$ be the solution created by the greedy algorithm.

Let’s show by induction that given any other optimal solution $H$, we can modify the first $k$ activities of $H$ so that they match the first $k$ activities of $G$, without introducing any overlap.

When $k = n$, the solution $H$ corresponds to $G$ and therefore $|G| = |H|$. 
Interval Scheduling

**Base Case:** $k = 1$

- Let the other optimal solution be $H = \{h_1, h_2, \ldots, h_m\}$
- We need to show that $g_1$ could substitute $h_1$
- By definition, we have that $f_{g_1} \leq f_{h_1}$
- Therefore, $g_1$ could stay in $h_1$ place without creating any overlap
- This proves that $g_1$ can be the beginning of any optimal solution!
**Inductive Step** (assuming it’s true until \( k \))

- We assume another optimal solution is \( H = \{g_1, \ldots, g_k, h_{k+1}, \ldots h_m\} \)
- We have to show that \( g_{k+1} \) could substitute \( h_{k+1} \)
- \( s_{g_{k+1}} \geq f_{g_k} \) (there is no overlap)
- Therefore, \( f_{g_{k+1}} \leq f_{h_{k+1}} \) (that is the way the greedy algorithm chooses)
- Given that, \( g_{k+1} \) could stay in \( h_{k+1} \)’s place without creating overlaps
- This proves that \( g_{k+1} \) could be chosen to extend our optimal (greedy) solution!
Greedy Algorithm for Interval Scheduling

- Sort the activities by increasing order of finish time
- Start by initializing $G = \emptyset$
- Keep adding to $G$ the next activity (that is, with the smaller $f_i$) that does not overlap with any activity of $G$

Temporal Complexity:
- Sort: $O(n \log n)$
- Process: $O(n)$
- Total: $O(n \log n)$
Imagine that summing $a$ and $b$ "costs" $a + b$. For instance, summing 1 with 10 would cost 11.

If we now want to sum the numbers $\{1, 2, 3\}$, there are several different ways (orders) to do it, giving origin to different total costs:

- **Hypothesis A**
  1. $1 + 2 = 3$ (cost 3)
  2. $3 + 3 = 6$ (cost 6)
  **Total Cost:** 9

- **Hypothesis B**
  1. $1 + 3 = 4$ (cost 3)
  2. $4 + 2 = 6$ (cost 6)
  **Total Cost:** 10

- **Hypothesis C**
  1. $2 + 3 = 5$ (cost 5)
  2. $5 + 1 = 6$ (cost 6)
  **Total Cost:** 11
Minimum Cost Sum

Minimum Cost Sum Problem

**Input:** a set of $n$ integer number.

**Output:** smallest possible total cost to sum them all, knowing that summing $a$ with $b$ costs $a + b$

Input/Output Example

**Input:** 3 numbers: \{1, 2, 3\}

**Input:** 9 (cost of summing $1 + 2$, followed by $3 + 3$).
**Minimum Cost Sum**

- What **greedy** would work here? And I’m already helping a lot by saying that greedy works...

- In each moment, choose the two smallest numbers in the set!

**Intuition:**

- the smaller the number, the smaller the cost
- the final sum is inevitable (whatever the order, it will have a cost equal to the sum of all numbers)
- consider the numbers $a$, $b$ and $c$. If the solution opts for $a + b$, that is because $c \geq a$ and $c \geq b$. Therefore, the cost of $a + c$ or $b + c$ would be higher or equal than $a + b$ and then we would have a sum of $a + b + c$.
- so, no other solution is better than our greedy choice!
Minimum Cost Sum

Greedy Algorithm for the Minimum Cost Sum

1. Repeat the following steps until the list of numbers becomes empty
   a. Remove the two smallest number $a$ and $b$
   b. Add $a + b$ to the total cost
   c. Insert $a + b$ to the list of numbers

Temporal Complexity:

- Number of Steps: $O(n)$
- Each step:
  a. Remove two smallest from list
  b. Add two numbers
  c. Insert one number in the list
- Total: Depends on the remove and insertion operations!
Minimum Cost Sum

- Add two numbers: $O(1)$!
- Insert and Remove:
  - Unordered array: insert in $O(1)$ and remove in $O(n)$
  - Ordered array: insert in $O(n)$ and remove in $O(1)$
  - Priority Queue (ex: heap): insert in $O(\log n)$ and remove in $O(\log n)$

Temporal Complexity:

- Number of Steps: $O(n)$
- Each step:
  - $O(n)$ for a "naive" algorithm
  - $O(\log n)$ if we use a specialized data structure
- Total: $O(n^2)$ or $O(n \log n)$
Greedy Algorithms

- A powerful and flexible idea

- The hard part is usually to prove that it gives an optimal result
  - Optimality is not guaranteed because we are not exploring completely all our search space
  - Generally it is easier to prove non-optimality (counter-example)
  - A simple way of analysing os to think about a case where there is a tie in the greedy condition: what would the algorithm choose in that case? Does it matter?

- When greedy works, usually it is efficient (low complexity)

- There is no “magic recipe” for all greedy algorithms: you need experience!