Optimization Problems

Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

Problem Instance: Input for the specified problem.

Problem Constraints: Requirements that must be satisfied by any feasible solution.

Feasible Solution: For any problem instance I, feasible(I) is the set of all solutions (i.e., outputs) for the instance I that satisfy the given constraints.

Objective Function: A function \( f: \text{feasible}(I) \rightarrow \mathbb{R} \cup \{0\} \). We often think of \( f \) as being a profit or a cost function.

Optimal Solution: A feasible solution \( X \in \text{feasible}(I) \) such that the profit \( f(X) \) is maximized (or the cost \( f(X) \) is minimized).

The Greedy Method

partial solutions

Given a problem instance \( I \), it should be possible to write a feasible solution \( X \) as a tuple \( \{x_1, x_2, \ldots, x_i\} \) for some integer \( i \), where \( x_i \subset X \) for all \( i \). A tuple \( \{x_1, x_2, \ldots, x_i\} \) where \( i \leq n \) is a partial solution if no constraints are violated.

Note: It may be the case that a partial solution cannot be extended to a feasible solution.

choice set

For a partial solution \( X = \{x_1, x_2, \ldots, x_i\} \) where \( i < n \), we define the choice set

\[ \text{choice}(X) = \{ y \in X : \{x_1, x_2, \ldots, x_i, y\} \text{ is a partial solution} \} \]
**CORE CHARACTERISTICS OF GREEDY ALGORITHMS**

Greedy algorithms do **not** look ahead and no backtracking.
Greedy algorithms can usually be implemented efficiently. Often they consist of a preprocessing step based on the function \( g \), followed by a single pass through the data.

In a greedy algorithm, only one feasible solution is constructed.
The execution of a greedy algorithm is based on local criteria (i.e., the values of the function \( g \)).

**Correctness.** For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!

**PROBLEM:** INTERVAL SELECTION

**POSSIBLE GREEDY STRATEGIES**

1. Sort the intervals in increasing order of **starting times**. At any stage, choose the earliest starting interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is \( s_i \)).
2. Sort the intervals in increasing order of **duration**. At any stage, choose the interval of **minimum duration** that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is \( f_i - s_i \)).
3. Sort the intervals in increasing order of **finishing time**. At any stage, choose the earliest finishing interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is \( f_i \)).

Does one of these strategies yield a correct greedy algorithm?

**STRATEGY 1: PROVING INCORRECTNESS**

- Idea: find one input for which the algorithm gives a non-optimal solution or an infeasible solution.

<table>
<thead>
<tr>
<th>Strategy 1</th>
<th>Consider input: ([0,10],[1,3],[5,7]).</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**POSSIBLE GREEDY STRATEGIES FOR INTERVAL SELECTION**

- Input: a set \( A = \{A_1, ..., A_n\} \) of time intervals
  - Each interval \( A_i \) has a start time \( s_i \) and a finish time \( f_i \).
- Feasible solution: a subset \( X \) of \( A \) containing pairwise disjoint intervals
- Output: a feasible solution of maximum size
  - i.e., one that maximizes \(|X|\)

- Chosen
- Rejected
- Not optimal

**WHERE** \( s_i \text{ and } f_i \) are **positive integers**.
2022-05-12

**Strategy 2**

Sort the intervals in increasing order of duration. At any stage, choose the interval of minimum duration that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).

Consider input:

\[(0, 5), [6, 10), [4, 7].\]

We will show that **Strategy 3** (sort in increasing order of finishing times) always yields the optimal solution.

---

**Strategy 3**

Time complexity:

Sort + one pass

\[\in \Theta(n \log n)\]

---

**How to prove** this is correct? (i.e., how can we show the returned solution is both feasible and optimal?)

**Feasibility?** Easy! We always choose an interval that starts after all other chosen intervals end.

**Optimality?** Harder...

---

**GREEDY CORRECTNESS PROOFS**

- Want to prove: greedy solution $X$ is correct (feasible & optimal).
- Usually show feasibility directly and optimality by contradiction:
  - Suppose solution $\mathcal{O}$ is better than $X$.
  - Show this necessarily leads to a contradiction.
- Two broad strategies for deriving this contradiction:
  1. **Greedy stays ahead**: show every choice in $X$ is “at least as good” as the corresponding choice in $\mathcal{O}$.
  2. **Exchange**: show $\mathcal{O}$ can be improved by replacing some choice in $\mathcal{O}$ with a choice in $X$.

Let’s demonstrate approach #1.

---

We give an induction proof.

Let $X$ be the greedy solution,

\[X = (A_i, \ldots, A_j),\]

where $i_1 < \cdots < i_k$.

Let $\mathcal{O}$ be any optimal solution,

\[\mathcal{O} = (A_{j_1}, \ldots, A_{j_l}),\]

where $j_1 < \cdots < j_k$.

We are merely imagining reordering the intervals chosen by the optimal algorithm so we can easily compare their finish times to intervals in $X$. 

---

**BRO I DON'T WANT PROOF**

I WANT EVIDENCE
A DIFFERENT PROOF

"Slick" ad-hoc approaches are sometimes possible...

Correctness Proof (cont.)

Recall: Greedy solution is \( X = (A_1, \ldots, A_m) \).
Optimal solution is \( O = (A_1, \ldots, A_p) \).
Now we complete the proof.
From the Lemma, we have \( f_m \leq f_m \).
Suppose that \( f > k \) to obtain a contradiction.
Let \( f' \) be the finishing time of the intervals in \( X \).
No interval finishes strictly to the left.
No interval starts strictly to the right.
No interval in \( X \) is strictly between these points!
So, in addition to the intervals in \( X \), only the following types of intervals are possible.
- Interval contains \( f_1 \)
- Interval contains \( f_2 \)
- Interval contains \( f_1 \) and \( f_2 \)
Thus, every interval contains some finishing time in \( F \).
And, two intervals in \( O \) cannot contain the same finishing time in \( F \).
So, there must be as many finishing times in \( F \) as there are intervals in \( O \).
QED

Problem: Interval Colouring

Instance: A set \( A = \{ A_1, \ldots, A_n \} \) of intervals.
For \( 1 \leq i \leq n \), \( A_i = (s_i, f_i) \), where \( s_i \) is the start time of interval \( A_i \) and \( f_i \) is the finish time of \( A_i \).
Feasible solution: A \( c \)-colouring is a mapping \( c : A \rightarrow \{1, \ldots, c\} \) that assigns each interval a colour such that two intervals receiving the same colour are always disjoint.
Find: A \( c \)-colouring of \( A \) with the minimum number of colours.

Example:

<table>
<thead>
<tr>
<th>Interval</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7 intervals. Feasible, but not optimal.
Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time. At a given point in time, suppose we have coloured the first $i < n$ intervals using $d$ colours. We will colour the $(i+1)$th interval with any permissible colour. If it cannot be coloured using any of the existing $d$ colours, then we introduce a new colour and $d$ is increased by 1. Question: In what order should we consider the intervals?

Consider intervals in the order they are given in the input: $A_1, \ldots, A_n$.
Can we do better?

Pre-sort intervals by increasing start time!
EXAMPLE: ORDER MATTERS!
Can we do better?

\[
\text{finish}_i = \text{finish time of last interval to receive colour } c_i
\]

Consider interval \( A_i = [s_i, f_i] \). If \( s_i \geq \text{finish}_i \), then we can give \( A_i \) colour \( c_i \) without breaking feasibility.

For each interval \( A_i \), search for an appropriate colour \( c \).

Initial state

\( i = 1 \)

Code before the loop: just assign colour 1.

\( d = 1 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!

\( i = 2 \)

\( d = 2 \)

\( \text{finish}[1] \)

Cannot reuse any colour. Create a new one!
Is $f_1 \leq s_3$? No. We cannot reuse colour 1.

While loop over $c$. Check if we can reuse a color in 1..d.

Is $f_2 \leq s_3$? No. We cannot reuse colour 2.

Cannot reuse any colour. Create new one.

Is $f_1 \leq s_4$? Yes. We can reuse colour 1.

Is $f_2 \leq s_4$? Yes. We can reuse colour 2.

Is $f_3 \leq s_4$? Yes. We can reuse colour 3.

Is $f_1 \leq s_5$? No. We cannot reuse colour 1.

Is $f_2 \leq s_5$? No. We cannot reuse colour 2.

Is $f_3 \leq s_5$? Yes. We can reuse colour 3.
Correctness of the Algorithm

The correctness of this greedy algorithm can be proven inductively as well as by a "slick" method—we give the "slick" proof:

Let \( D \) denote the number of colours used by the algorithm.

Let \( F_D \) be the first interval that has colour \( D \)
Let \( L_c \) be the last interval that has colour \( c \) and starts before \( F_D \) ends
we prove \( F_D \) overlaps every interval \( L_c \) for all \( c < D \)

If \( D = 1 \) the proof is immediate, so suppose \( D \geq 2 \)

Let's argue \( L_1 \) overlaps \( F_D \)
Note \( L_1 \) must exist (otherwise greedy would use colour 1 for \( F_D \))
And \( f_i \) must be after \( F_D \) starts (same reason)
So \( L_1 \) finishes during \( F_D \)
Some argument applies during \( F_D \)
So, \( F_D \) overlaps \( D - 1 \) intervals!
Moreover, every interval in \( \{L_1, \ldots, L_{D-1}\} \) contains the starting time of \( F_D \)
So, we must use \( D \) colours!

IMPROVING THIS ALGORITHM

- Current greedy algorithm:
  - For each interval \( A_c \), compare its start time \( s_i \) with the \( \text{finish}[c] \) times of all colours introduced so far.
  - Why? Looking for some \( \text{finish}[c] \) time that is earlier than \( s_i \)
  - We are doing linear search... Can we do better?
  - Use a priority queue to keep track of the earliest \( \text{finish}[c] \) at all times in the algorithm.
  - Then we only need to look at minimum element
EXAMPLE: HEAP-BASED ALGORITHM

Initial state
Min element: NULL
Heap:

Iteration 1
Check heap minimum
Empty, so a new colour is needed

Iteration 2
Check heap minimum
Check if finish time 3 is before $s_2$
No. New colour!

Iteration 3
Check heap minimum
Check if finish time 3 is before $s_3$
No. New colour!

Iteration 4
Check heap minimum
Check if finish time 7 is before $s_4$
No. New colour!
**Example: Heap-Based Algorithm**

**Iteration 3**
- Check heap minimum
- Check if finish time \( s_3 \) is before \( s_4 \)
- No. New colour!

**Iteration 4**
- Check heap minimum
- Check if finish time \( s_3 \) is before \( s_4 \)
- Yes. Reuse colour, deleteMin and insert new finish time into heap!
Heap-based algorithm:

- **Iteration 5:**
  - Min element: \( A_7 \)
  - Check heap minimum, check if finish time 7 is before \( s_5 \), reuse, deleteMin, and insert new finish time into heap.

- **Iteration 6:**
  - Min element: \( A_8 \)
  - Check heap minimum, check if finish time 7 is before \( s_6 \), reuse, deleteMin, and insert new finish time into heap.

- **Iteration 7:**
  - Min element: \( A_9 \)
  - Check heap minimum, check if finish time 8 is before \( s_7 \), reuse, deleteMin, and insert new finish time into heap.

**Complexity Analysis:**

- \( O(1) \)
- \( O(\log D) \)
- \( O(\log S) \)

where \( S = \) size of priority queue.

**Total complexity:**

\[ \Theta(n \log n) + \Theta(n \log D) \]

Since \( n \geq D \), \( \Theta(n \log D) \).