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., solutions) for the instance $I$ that satisfy the given constraints.
- **Objective Function**: A function $f : \text{feasible}(I) \rightarrow \mathbb{R}^+$, i.e., $f(i)$. 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 Solution**: 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 $x_i \in X$ for all $i$. A tuple $(x_1, x_2, \ldots, x_i)$ where $i < n$ 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 $X = (x_1, \ldots, x_i)$ where $i < n$, we define the choice set $\text{choice}(X) = \{ y \in X : (x_1, \ldots, x_i, y) \text{ is a partial solution} \}$.

The Greedy Method (cont.)

- **Local Evaluation Criterion**: For any $y \in X$, $g(y)$ is a local evaluation criterion that measures the cost or profit of including $y$ in a (partial) solution.
- **Extension**: Given a partial solution $X = (x_1, \ldots, x_i)$, where $i < n$, choose $y \in \text{choice}(X)$ so that $g(y)$ is as small (or large) as possible. Update $X$ to be the $(i + 1)$-tuple $(x_1, \ldots, x_i, y)$.
- **Greedy Algorithm**: Starting with the “empty” partial solution, repeatedly extend it until a feasible solution $X$ is constructed. This feasible solution may or may not be optimal.

SOLVING OPTIMIZATION PROBLEMS

- **Lots of Techniques**
- **We will study greedy approaches first**
- **Later, dynamic programming**
- **Sort of like divide and conquer**
- **But can sometimes be much more efficient than D&C**
- **Greedy algorithms are usually**
- **Very fast, but hard to prove optimality for**
- **Structured as follows...**
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 $\mathcal{g}$, 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 $\mathcal{g}$).

**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

**Input:** a set $A = \{A_1, \ldots, A_n\}$ 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|

### POSSIBLE GREEDY STRATEGIES

- **Partial solutions**
  - $X = \{x_1, x_2, \ldots, x_l\}$, where each $x_i$ is an interval for the output
- **Choices**
  - $X = A$ (i.e., all intervals)
  - Choice($X$) = $\{y \in X : \{x_1, \ldots, x_l, y\}$ respects all constraints $\}$
  - i.e., where $y \notin X$ and $X \cup y$ disjoint($y, x$)
- **Local evaluation function**
  - $g(y) = s_i$ where $y = A_i$
  - i.e., $g(y) =$ start time of interval $y$

### 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:</th>
<th>0, 10), (1, 3), (5, 7).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strategy 1</strong></td>
<td></td>
<td>0 2 4 6 8 10 x-axis</td>
</tr>
</tbody>
</table>
**策略2**

考虑输入：\((4, 5), (6, 30), (4, 7)\)。

我们将显示，策略3（按完成时间降序）总是产生最优解。

**策略3**

如何证明这是正确的？（即，我们如何证明返回的解决方案是可行的？）

**可行性？**容易！我们总是选择一个在所有已选区间之后开始的区间。

**优化？**更难……

**贪心正确性证明**

- 要证明：贪婪解\(X\)是正确的（可行且最优）。
- 通常直接证明可行性和通过反证证明最优性。
- 假设解决方案0比\(X\)好。
- 显示这必然导致矛盾。
- 两种方法来推导这个矛盾：
  1. **贪婪提前**：显示\(X\)中的每一个选择都是“至少和好”。
  2. **交换**：显示\(O\)可以通过替换\(O\)中的某些选择来改进，而这些选择在\(X\)中。

### 贪心贪心

我们给出一个归纳证明。

让\(X\)是贪婪解：

\[ X = (X_1, \ldots, X_k) \]

其中

\[ i_1 < \cdots < i_k \]

让\(O\)是任何最优解：

\[ O = (O_1, \ldots, O_m) \]

其中

\[ j_1 < \cdots < j_m \]

我们只是假定，我们没有重新排序共同选择的区间，而是简单地将它们的完成时间与\(X\)中的区间进行比较。
A DIFFERENT PROOF

“Slick” ad-hoc approaches are sometimes possible...

**Problem: Interval Colouring**

**Instance:** A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ 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 $\sigma : \mathcal{A} \rightarrow \{1, \ldots, c\}$ that assigns each interval a colour such that two intervals receiving the same colour are always disjoint.

**Find:** A $\sigma$-colouring of $\mathcal{A}$ with the minimum number of colours.

**Example:** A feasible, but not optimal $\sigma$-colouring.
We will colour the \((i + 1)\)st 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.

**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 \((i + 1)\)st 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?
Can we do better?
Can we do better?

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

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

For each interval \( A_i \), search for an appropriate colour \( c \).
If we didn't reuse a colour, we can create a new one!

Initial state

---

**EXAMPLE: ORDER MATTERS!**

Used 3 colours

**EXAMPLE: RUNNING GREEDY**

**EXAMPLE: RUNNING GREEDY**

**EXAMPLE: RUNNING GREEDY**

**EXAMPLE: RUNNING GREEDY**
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.

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

Is $f_2 \leq s_4$? No. We cannot 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.

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

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—let’s 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 $L_1$ 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 just use colour 1 for $F_D$).
And $f_{\text{finish}}[L_1]$ must be after $F_D$ starts (same reason).
So $L_1$ finishes (same reason).
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!

TIME COMPLEXITY?

- Total $\Theta(n \log n)$ if only a constant number of colours are needed.
- Could be $\Theta(n \log n)$ if only a constant number of colours are needed.
- Could be $\Theta(n^2)$ if $n$ colours are needed.
- Most accurate complexity statement is $\Theta(n \log n + nd)$ where $d$ is the number of colours used.

What inefficiencies exist in this algorithm? Could we make it faster with clever data structure usage?

IMPROVING THIS ALGORITHM

- Current greedy algorithm:
  - For each interval $A_i$, compare its start time $s_i$ with the finish times of all $n$ colours.
- Why? Looking for the earliest finish 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 finish times at all times in the algorithm.
- Then we only need to look at the minimum element.
**EXAMPLE: HEAP-BASED ALGORITHM**

**Initial state**

Min element: NULL

Heap:

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Check heap minimum**

Empty, so a new colour is needed

**Iteration \(i = 1\)**

Finish at time 3

**Iteration \(i = 2\)**

Finish at time 7

**Iteration \(i = 3\)**

Finish at time 7

**Check if finish time is before \(s_2\)**

No new colour!
EXAMPLE: HEAP-BASED ALGORITHM

Iteration: 3
Check heap minimum: 3 is before $s_3$

No. New colour!

Iteration: 4
Check heap minimum: 3 is before $s_4$

Yes. Reuse colour, deleteMin and insert new finish time into heap!

EXAMPLE: HEAP-BASED ALGORITHM

Iteration: 4
Check heap minimum: 3 is before $s_4$

Yes. Reuse colour, deleteMin and insert new finish time into heap!

EXAMPLE: HEAP-BASED ALGORITHM

Iteration: 5
Check heap minimum: 5 is before $s_5$

Yes. Reuse colour, deleteMin and insert new finish time into heap!

EXAMPLE: HEAP-BASED ALGORITHM

Iteration: 5
Check heap minimum: 5 is before $s_5$

Yes. Reuse colour, deleteMin and insert new finish time into heap!

EXAMPLE: HEAP-BASED ALGORITHM

Iteration: 5
Check heap minimum: 5 is before $s_5$

Yes. Reuse colour, deleteMin and insert new finish time into heap!
Check heap minimum

Heap Min element: NULL

Check if finish time 5 is before $s_5$

Iteration $i = 5$

Finish at time 7

Finish at time 9

Yes. Reuse colour, deleteMin and insert new finish time into heap!

Finish at time 7

Finish at time 13

And so on, and so forth...

$O(1)$ where $f = \text{size(priority queue)}$

$O(1)$

$O(1)$

$O(\log f)$

$O(\log f)$

$O(\log f)$

$O(\log n)$

$O(n \log n)$ + $O(n \log D)$

Since $n \geq D$, $O(n \log D)$. Total $O(n \log n) + O(n \log D)$

Preprocess: A[1][0]

Sort A by increasing start time

Let $s[i]$ be the start times in A

Let $f[i]$ be the finish time in A

Return GreedyIntervalColouring($s$, $f$)

GreedyIntervalColouring($s$, $f$)

$\text{colour}[i] = \{\}$

$\text{h.insert}([f[i], \text{colour}[i]])$

for $i = 0$ to $n - 1$

if $f[i] < s[i]$

then

$\text{h.deleteMin}$

else

$\text{colour}[i] = \text{h.min}$

$\text{h.insert}([f[i], \text{colour}[i]])$

return $\text{colour}$