CS498: Algorithmic Engineering

Lecture 6: Modeling Patterns for Integer Programs

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 03 – 02/05/2026

Outline

0–1 Knapsack and Its LP Relaxation

General 0–1 knapsack: $\max \sum_{i=1}^{n} v_i x_i$

$$\text{s.t. } \sum_{i=1}^{n} w_i x_i \le C, \quad x_i \in \{0, 1\} \text{ for all } i.$$

0–1 Knapsack and Its LP Relaxation

General 0–1 knapsack: $\max \sum_{i=1}^{n} v_i x_i$

$$\text{s.t. } \sum_{i=1}^{n} w_i x_i \leq C, \quad x_i \in \{0,1\} \text{ for all } i.$$

LP relaxation:

$$0 \leq x_i \leq 1 \quad \text{instead of } x_i \in \{0,1\}.$$

0–1 Knapsack and Its LP Relaxation

General 0–1 knapsack: $\max \sum_{i=1}^{n} v_i x_i$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq C, \quad x_i \in \{0, 1\} \text{ for all } i.$$

LP relaxation:

$$0 \leq x_i \leq 1 \quad \text{instead of } x_i \in \{0, 1\}.$$

Now the LP is allowed to take *fractions* of items:

$$x_1 = 1,\ x_2 = 0.4,\ x_3 = 0.6, \ldots$$

0–1 Knapsack and Its LP Relaxation

General 0–1 knapsack: $\max \sum_{i=1}^{n} v_i x_i$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le C, \quad x_i \in \{0,1\} \text{ for all } i.$$

LP relaxation:

$$0 \le x_i \le 1 \quad \text{instead of } x_i \in \{0,1\}.$$

Now the LP is allowed to take *fractions* of items:

$$x_1 = 1,\ x_2 = 0.4,\ x_3 = 0.6, \ldots$$

At the root node of B&B:

- LP value at root becomes the upper bound $UB^{(0)}_{\text{root}}$.

0–1 Knapsack and Its LP Relaxation

General 0–1 knapsack: $\max \sum_{i=1}^{n} v_i x_i$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le C, \quad x_i \in \{0,1\} \text{ for all } i.$$

LP relaxation:

$$0 \le x_i \le 1 \quad \text{instead of } x_i \in \{0,1\}.$$

Now the LP is allowed to take *fractions* of items:

$$x_1 = 1,\ x_2 = 0.4,\ x_3 = 0.6, \ldots$$

At the root node of B&B:

- LP value at root becomes the upper bound $UB^{(0)}_{\text{root}}$.
- Any integer solution has value $\le z^*_{\text{ILP}}$.

0–1 Knapsack and Its LP Relaxation

General 0–1 knapsack: $\max \sum_{i=1}^{n} v_i x_i$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq C, \quad x_i \in \{0,1\} \text{ for all } i.$$

LP relaxation:

$$0 \leq x_i \leq 1 \quad \text{instead of } x_i \in \{0,1\}.$$

Now the LP is allowed to take *fractions* of items:

$$x_1 = 1,\ x_2 = 0.4,\ x_3 = 0.6, \ldots$$

At the root node of B&B:

- LP value at root becomes the upper bound $UB^{(0)}_{\text{root}}$.
- Any integer solution has value $\leq z^*_{\text{ILP}}$.

Typically: $UB^{(0)}_{\text{root}} > z^*_{\text{ILP}}$, so the solver *must* branch to prove there is nothing better than z^*_{ILP}.

Strengthening with Cover Inequalities

A **cover** $S \subseteq \{1, \ldots, n\}$ is any set of items with $\sum_{i \in S} w_i > C$.
These items **do not all fit** in the knapsack.

Strengthening with Cover Inequalities

A **cover** $S \subseteq \{1, \ldots, n\}$ is any set of items with $\sum_{i \in S} w_i > C$.
These items **do not all fit** in the knapsack. We can build stronger and stronger LP relaxations:

Strengthening with Cover Inequalities

A **cover** $S \subseteq \{1, \ldots, n\}$ is any set of items with $\sum_{i \in S} w_i > C$.
These items **do not all fit** in the knapsack. We can build stronger and stronger LP relaxations:

- **Model** M_0 **(weak):** only the knapsack constraint, $\sum_{i=1}^{n} w_i x_i \leq C$.

Strengthening with Cover Inequalities

A **cover** $S \subseteq \{1, \dots, n\}$ is any set of items with $\sum_{i \in S} w_i > C$.

These items **do not all fit** in the knapsack. We can build stronger and stronger LP relaxations:

- **Model M_0 (weak):** only the knapsack constraint, $\sum_{i=1}^{n} w_i x_i \leq C$.
- **Model M_1 (pairs):** M_0 + all 2-item covers

$$x_i + x_j \leq 1 \quad \text{for every pair } \{i, j\} \text{ with } w_i + w_j > C.$$

Strengthening with Cover Inequalities

A **cover** $S \subseteq \{1, \ldots, n\}$ is any set of items with $\sum_{i \in S} w_i > C$.
These items **do not all fit** in the knapsack. We can build stronger and stronger LP relaxations:

- **Model M_0 (weak):** only the knapsack constraint, $\sum_{i=1}^{n} w_i x_i \leq C$.
- **Model M_1 (pairs):** M_0 + all 2-item covers

$$x_i + x_j \leq 1 \quad \text{for every pair } \{i, j\} \text{ with } w_i + w_j > C.$$

- **Model M_2 (triples):** M_1 + all 3-item covers

$$x_i + x_j + x_k \leq 2 \quad \text{for every triple with } w_i + w_j + w_k > C.$$

Strengthening with Cover Inequalities

A **cover** $S \subseteq \{1, \ldots, n\}$ is any set of items with $\sum_{i \in S} w_i > C$.
These items **do not all fit** in the knapsack. We can build stronger and stronger LP relaxations:

- **Model M_0 (weak):** only the knapsack constraint, $\sum_{i=1}^{n} w_i x_i \leq C$.
- **Model M_1 (pairs):** M_0 + all 2-item covers

$$x_i + x_j \leq 1 \quad \text{for every pair } \{i, j\} \text{ with } w_i + w_j > C.$$

- **Model M_2 (triples):** M_1 + all 3-item covers

$$x_i + x_j + x_k \leq 2 \quad \text{for every triple with } w_i + w_j + w_k > C.$$

- ...and so on, adding covers of size $4, 5, \ldots, k$.

Each step: same integer problem, but LP relaxation gets tighter.

How the Number of Inequalities Blows Up

For each cover S we add

$$\sum_{i \in S} x_i \leq |S| - 1.$$

How the Number of Inequalities Blows Up

For each cover S we add

$$\sum_{i \in S} x_i \leq |S| - 1.$$

Counting them:

- All 2-item covers: up to $\binom{n}{2} = O(n^2)$ inequalities.

How the Number of Inequalities Blows Up

For each cover S we add

$$\sum_{i \in S} x_i \leq |S| - 1.$$

Counting them:

- All 2-item covers: up to $\binom{n}{2} = O(n^2)$ inequalities.
- All 3-item covers: up to $\binom{n}{3} = O(n^3)$ inequalities.

How the Number of Inequalities Blows Up

For each cover S we add

$$\sum_{i \in S} x_i \leq |S| - 1.$$

Counting them:

- All 2-item covers: up to $\binom{n}{2} = O(n^2)$ inequalities.
- All 3-item covers: up to $\binom{n}{3} = O(n^3)$ inequalities.
- All covers with $|S| \leq k$: up to $O(n^k)$ inequalities.
- All covers of any size: potentially exponentially many.

How the Number of Inequalities Blows Up

For each cover S we add

$$\sum_{i \in S} x_i \leq |S| - 1.$$

Counting them:

- All 2-item covers: up to $\binom{n}{2} = O(n^2)$ inequalities.
- All 3-item covers: up to $\binom{n}{3} = O(n^3)$ inequalities.
- All covers with $|S| \leq k$: up to $O(n^k)$ inequalities.
- All covers of any size: potentially exponentially many.

As we move from M_0 to M_1 to M_2 to . . . :

$$z^*_{\mathrm{LP}(M_0)} \geq z^*_{\mathrm{LP}(M_1)} \geq z^*_{\mathrm{LP}(M_2)} \geq \cdots \geq z^*_{\mathrm{ILP}}.$$

LP bounds get closer to the true integer optimum,

How the Number of Inequalities Blows Up

For each cover S we add

$$\sum_{i \in S} x_i \leq |S| - 1.$$

Counting them:

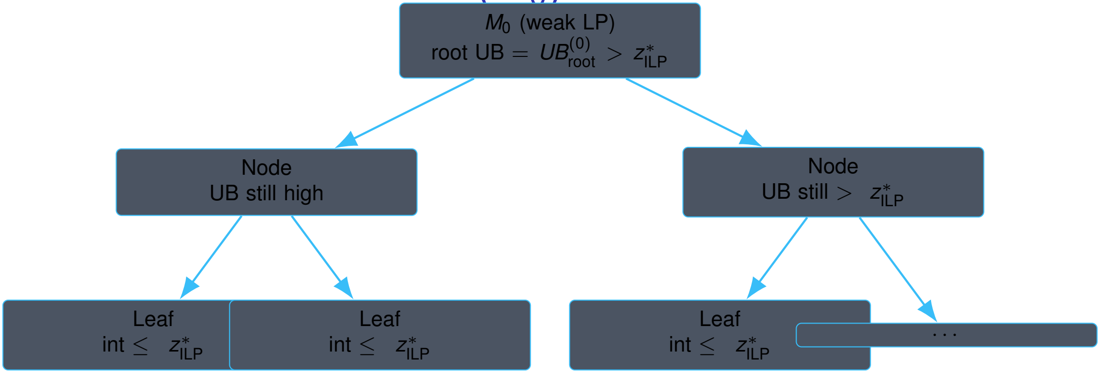
- All 2-item covers: up to $\binom{n}{2} = O(n^2)$ inequalities.
- All 3-item covers: up to $\binom{n}{3} = O(n^3)$ inequalities.
- All covers with $|S| \leq k$: up to $O(n^k)$ inequalities.
- All covers of any size: potentially exponentially many.

As we move from M_0 to M_1 to M_2 to $\ldots$:

$$z^*_{\mathrm{LP}(M_0)} \;\geq\; z^*_{\mathrm{LP}(M_1)} \;\geq\; z^*_{\mathrm{LP}(M_2)} \;\geq\; \cdots \;\geq\; z^*_{\mathrm{ILP}}.$$

LP bounds get closer to the true integer optimum,
But the LP itself gets larger and more expensive to solve (more rows, denser constraint matrix).

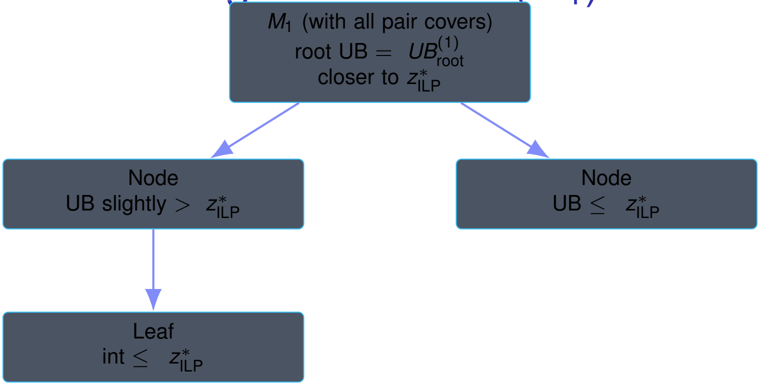
Effect on B&B: Weak LP (M_0)



Behavior:

- Root UB far above the best integer value.
- LP relaxation gives weak guidance.
- Solver must explore many nodes before pruning.

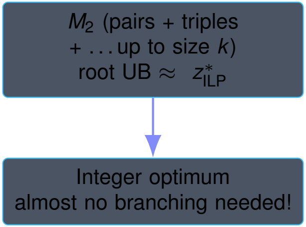
Effect on B&B: Adding Pair Covers (M_1)



Behavior:

- Pair-cover inequalities tighten the relaxation.
- Root UB moves closer to the true integer optimum.
- Some nodes are pruned much earlier than in M_0.

Effect on B&B: Adding Higher-Order Covers (M_2)



Behavior:

- Root LP bound is very close to, or equal to, z^*_{ILP}.
- The B&B tree essentially collapses to the root.
- But the LP now has $O(n^k)$ extra inequalities. Solving the LP takes as much as brute forcing!

Formulation Strength vs Search Effort

- Starting from the basic knapsack LP (M_0), bounds are loose, and the solver has to explore many nodes.
- Adding all pair covers (M_1) and then all triples, etc. (M_2) makes the LP bounds tighter and the B&B tree smaller.

Formulation Strength vs Search Effort

- Starting from the basic knapsack LP (M_0), bounds are loose, and the solver has to explore many nodes.
- Adding all pair covers (M_1) and then all triples, etc. (M_2) makes the LP bounds tighter and the B&B tree smaller.

But:

- The number of cover inequalities grows as $O(n^2)$, $O(n^3)$, $\ldots$, $O(n^k)$.
- Solving the LP at each node becomes more and more expensive.

Formulation Strength vs Search Effort

- Starting from the basic knapsack LP (M_0), bounds are loose, and the solver has to explore many nodes.
- Adding all pair covers (M_1) and then all triples, etc. (M_2) makes the LP bounds tighter and the B&B tree smaller.

But:

- The number of cover inequalities grows as $O(n^2), O(n^3), \dots, O(n^k)$.
- Solving the LP at each node becomes more and more expensive.

Big picture:

$$\text{Stronger LP} \;\Rightarrow\; \text{fewer nodes but bigger LPs.}$$

Good models strike a balance:

- LP strong enough to give meaningful bounds,
- but not so huge that the LP solve itself dominates the running time.

Why Modeling Patterns Matter

- In practice, the hard part is rarely "solving" the IP.

Why Modeling Patterns Matter

- In practice, the hard part is rarely “solving” the IP.
- The hard part is **formulating** the problem so that:
  - it captures the real-world constraints, and
  - it is solver-friendly (strong LP relaxation).

Why Modeling Patterns Matter

- In practice, the hard part is rarely “solving” the IP.
- The hard part is **formulating** the problem so that:
  - it captures the real-world constraints, and
  - it is solver-friendly (strong LP relaxation).
- Instead of reinventing the wheel every time, we reuse patterns:
  - selection / knapsack,
  - fixed-charge on/off,
  - logical implications,
  - cardinality and “at most k”,
  - either–or and disjunctions,
  - SOS1/SOS2 for piecewise-linear functions.

Why Modeling Patterns Matter

- In practice, the hard part is rarely "solving" the IP.
- The hard part is **formulating** the problem so that:
  - it captures the real-world constraints, and
  - it is solver-friendly (strong LP relaxation).
- Instead of reinventing the wheel every time, we reuse patterns:
  - selection / knapsack,
  - fixed-charge on/off,
  - logical implications,
  - cardinality and "at most k",
  - either–or and disjunctions,
  - SOS1/SOS2 for piecewise-linear functions.

Today's lecture = "design patterns" for integer programming.

Pattern 1: Subset Selection / 0–1 Knapsack

Basic 0–1 knapsack:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \sum_{i=1}^{n} w_i x_i \le C, \quad x_i \in \{0, 1\}.$$

Pattern 1: Subset Selection / 0–1 Knapsack

Basic 0–1 knapsack:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq C, \quad x_i \in \{0, 1\}.$$

Interpretation:

- $x_i = 1 \Rightarrow$ choose item i.
- $x_i = 0 \Rightarrow$ do not choose item i.

Pattern 1: Subset Selection / 0–1 Knapsack

Basic 0–1 knapsack:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \sum_{i=1}^{n} w_i x_i \le C, \quad x_i \in \{0, 1\}.$$

Interpretation:

- $x_i = 1 \Rightarrow$ choose item i.
- $x_i = 0 \Rightarrow$ do not choose item i.

Pattern name: binary **subset selection**.

Pattern 2: Fixed-Charge / On–Off Decisions

Many problems have a **fixed cost** to “turn something on” before you use it:

- Opening a facility.
- Starting up a machine for the day.
- Using a particular shipping route (e.g. Suez Canal registration fees).

Pattern 2: Fixed-Charge / On–Off Decisions

Many problems have a **fixed cost** to “turn something on” before you use it:

- Opening a facility.
- Starting up a machine for the day.
- Using a particular shipping route (e.g. Suez Canal registration fees).

Variables:

- $y \in \{0, 1\}$: on/off decision.
- $q \geq 0$: flow / production / quantity.

Pattern 2: Fixed-Charge / On–Off Decisions

Many problems have a **fixed cost** to "turn something on" before you use it:

- Opening a facility.
- Starting up a machine for the day.
- Using a particular shipping route (e.g. Suez Canal registration fees).

Variables:

- $y \in \{0, 1\}$: on/off decision.
- $q \geq 0$: flow / production / quantity.

Model: $0 \leq q \leq Q_{\max} y$ and $y \in \{0, 1\}$.

Pattern 2: Fixed-Charge / On–Off Decisions

Many problems have a **fixed cost** to "turn something on" before you use it:

- Opening a facility.
- Starting up a machine for the day.
- Using a particular shipping route (e.g. Suez Canal registration fees).

Variables:

- $y \in \{0, 1\}$: on/off decision.
- $q \geq 0$: flow / production / quantity.

Model: $0 \leq q \leq Q_{\text{max}} y$ and $y \in \{0, 1\}$.
If $y = 0 \Rightarrow q = 0$
if $y = 1 \Rightarrow 0 \leq q \leq Q_{\text{max}}$

Fixed-Charge Example: Mini Factory Startup

Simple factory:

- Revenue: \$6 per unit produced.
- Fixed startup cost: \$200 if we open.
- Capacity: at most 50 units.

Fixed-Charge Example: Mini Factory Startup

Simple factory:

- Revenue: \$6 per unit produced.
- Fixed startup cost: \$200 if we open.
- Capacity: at most 50 units.

$$\begin{aligned} \max\ & 6q - 200y \\ \text{s.t. } & 0 \le q \le 50y, \\ & y \in \{0, 1\}. \end{aligned}$$

Fixed-Charge Example: Mini Factory Startup

Simple factory:

- Revenue: \$6 per unit produced.
- Fixed startup cost: \$200 if we open.
- Capacity: at most 50 units.

$$
\begin{aligned}
\max\ & 6q - 200y \\
\text{s.t. } & 0 \leq q \leq 50y, \\
& y \in \{0, 1\}.
\end{aligned}
$$

Observations:

- If $y = 0$: we get $q = 0$, profit $= 0$.
- If $y = 1$: we choose $q = 50$.

Fixed-Charge Example: Mini Factory Startup

Simple factory:

- Revenue: \$6 per unit produced.
- Fixed startup cost: \$200 if we open.
- Capacity: at most 50 units.

$$\max\ 6q - 200y$$
$$\text{s.t. } 0 \leq q \leq 50y,$$
$$y \in \{0, 1\}.$$

Observations:

- If $y = 0$: we get $q = 0$, profit $= 0$.
- If $y = 1$: we choose $q = 50$.

Pattern name: *fixed-charge on/off*.

Pattern 3: Logical Implications (If A then B)

Suppose we have 2 options:

$$A, B \in \{0, 1\}.$$

Requirements:

- If A is chosen, B must be chosen.

Pattern 3: Logical Implications (If A then B)

Suppose we have 2 options:

$$A, B \in \{0, 1\}.$$

Requirements:

- If A is chosen, B must be chosen.

Encoding:

$$A \leq B.$$

Pattern 3: Logical Implications (If A then B)

Suppose we have 2 options:

$$A, B \in \{0, 1\}.$$

Requirements:

- If A is chosen, B must be chosen.

Encoding:

$$A \leq B.$$

- If $A = 1$ then $B = 1$.
- If $A = 0$ then $B \in \{0, 1\}$.

Pattern 3: Logical Implications (If A then B)

Suppose we have 2 options:

$$A, B \in \{0, 1\}.$$

Requirements:

- If A is chosen, B must be chosen.

Encoding:

$$A \leq B.$$

- If $A = 1$ then $B = 1$.
- If $A = 0$ then $B \in \{0, 1\}$.

Pattern name: *binary implication / precedence constraints*.

Pattern 3 in Gurobi: Logical Implication with >>

Goal: If $A = 1$, then enforce constraint $a^\top x \leq b$.

```
import gurobipy as gp
from gurobipy import GRB

m = gp.Model("implication_demo")

# Binary "trigger" variable A
A = m.addVar(vtype=GRB.BINARY, name="A")

# Some continuous variables x
x1 = m.addVar(lb=0.0, name="x1")
x2 = m.addVar(lb=0.0, name="x2")

# If A == 1, then x1 + 2*x2 <= 5
m.addConstr((A == 1) >> (x1 + 2*x2 <= 5), name="A_implies_constr") #Only new thing here.
m.setObjective(x1 + x2 - 10*A, GRB.MINIMIZE)
m.optimize()
```

Notes: The constraint only *activates* when $A = 1$.

Pattern 4: Either–Or Constraints

Many decisions are mutually exclusive:

- Route A or Route B (but not both).
- Use technology 1 or technology 2.
- Choose one of several pricing schemes.

Pattern 4: Either–Or Constraints

Many decisions are mutually exclusive:

- Route A or Route B (but not both).
- Use technology 1 or technology 2.
- Choose one of several pricing schemes.

Basic idea: introduce binaries z_A, z_B:

$$z_A + z_B = 1, \quad z_A, z_B \in \{0, 1\}.$$

Pattern 4: Either–Or Constraints

Many decisions are mutually exclusive:

- Route A or Route B (but not both).
- Use technology 1 or technology 2.
- Choose one of several pricing schemes.

Basic idea: introduce binaries z_A, z_B:

$$z_A + z_B = 1, \quad z_A, z_B \in \{0, 1\}.$$

Then “gate” each option:

- If Route A: constraints for Route A active when $z_A = 1$.
- If Route B: constraints for Route B active when $z_B = 1$.

Either–Or Example: Two Shipping Routes

Demand: D units, must ship all of it. Two routes:

- Route 1: cost c_1 per unit, capacity U_1.
- Route 2: cost c_2 per unit, capacity U_2.

Either–Or Example: Two Shipping Routes

Demand: D units, must ship all of it. Two routes:

- Route 1: cost c_1 per unit, capacity U_1.
- Route 2: cost c_2 per unit, capacity U_2.

Variables:

$$f_1, f_2 \geq 0, \quad z_1, z_2 \in \{0, 1\}.$$

Either–Or Example: Two Shipping Routes

Demand: D units, must ship all of it. Two routes:

- Route 1: cost c_1 per unit, capacity U_1.
- Route 2: cost c_2 per unit, capacity U_2.

Variables:

$$f_1, f_2 \geq 0, \quad z_1, z_2 \in \{0, 1\}.$$

Constraints:

$$f_1 + f_2 = D,$$
$$f_1 \leq U_1 z_1, \quad f_2 \leq U_2 z_2,$$
$$z_1 + z_2 = 1.$$

Objective:

$$\min \; c_1 f_1 + c_2 f_2.$$

Either–Or Example: Two Shipping Routes

Demand: D units, must ship all of it. Two routes:

- Route 1: cost c_1 per unit, capacity U_1.
- Route 2: cost c_2 per unit, capacity U_2.

Variables:

$$f_1, f_2 \geq 0, \quad z_1, z_2 \in \{0, 1\}.$$

Constraints:

$$f_1 + f_2 = D,$$
$$f_1 \leq U_1 z_1, \quad f_2 \leq U_2 z_2,$$
$$z_1 + z_2 = 1.$$

Objective:

$$\min \; c_1 f_1 + c_2 f_2.$$

Pattern name: *either–or / disjunctive constraints.*

Pattern 5: Big-M

We often need a binary variable z to indicate if a linear inequality is satisfied.

$$z = 1 \implies a^\top x \leq b$$

Pattern 5: Big-M

We often need a binary variable z to indicate if a linear inequality is satisfied.

$$z = 1 \implies a^\top x \leq b$$

Using Big-M, we can encode this logic linearly:

$$a^\top x \leq b + M(1 - z) \qquad z \in \{0, 1\}$$

Pattern 5: Big-M

We often need a binary variable z to indicate if a linear inequality is satisfied.

$$z = 1 \implies a^\top x \leq b$$

Using Big-M, we can encode this logic linearly:

$$a^\top x \leq b + M(1 - z) \qquad z \in \{0, 1\}$$

- If $z = 1$: constraint is enforced $\Rightarrow a^\top x \leq b$.

Pattern 5: Big-M

We often need a binary variable z to indicate if a linear inequality is satisfied.

$$z = 1 \implies a^\top x \leq b$$

Using Big-M, we can encode this logic linearly:

$$a^\top x \leq b + M(1 - z) \qquad z \in \{0, 1\}$$

- If $z = 1$: constraint is enforced $\Rightarrow a^\top x \leq b$.
- If $z = 0$: constraint relaxed $\Rightarrow a^\top x \leq b + M$ (always true if M large enough).

Pattern 5: Big-M

We often need a binary variable z to indicate if a linear inequality is satisfied.

$$z = 1 \implies a^\top x \leq b$$

Using Big-M, we can encode this logic linearly:

$$a^\top x \leq b + M(1 - z) \qquad z \in \{0, 1\}$$

- If $z = 1$: constraint is enforced $\Rightarrow a^\top x \leq b$.
- If $z = 0$: constraint relaxed $\Rightarrow a^\top x \leq b + M$ (always true if M large enough).

Question: How do we choose M?

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \le y \le 1$) of:

$$\min y \text{ s.t.} \quad x \ge 1, \qquad x \le My, \qquad y \in \{0, 1\}.$$

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \le y \le 1$) of:

$$\min y \text{ s.t. } \quad x \ge 1, \qquad x \le My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \ge 1$, $x \le y \Rightarrow y \ge 1$.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \leq y \leq 1$) of:

$$\min y \text{ s.t. } \quad x \geq 1, \qquad x \leq My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \geq 1$, $x \leq y \Rightarrow y \geq 1$.
- Feasible LP region: only $x = 1$, $y = 1$.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \le y \le 1$) of:

$$\min y \text{ s.t. } \quad x \ge 1, \qquad x \le My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \ge 1$, $x \le y \Rightarrow y \ge 1$.
- Feasible LP region: only $x = 1$, $y = 1$.
- LP relaxation = integer solution $\implies$ **strong LP**.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \leq y \leq 1$) of:

$$\min y \text{ s.t. } \quad x \geq 1, \qquad x \leq My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \geq 1$, $x \leq y \Rightarrow y \geq 1$.
- Feasible LP region: only $x = 1$, $y = 1$.
- LP relaxation = integer solution $\implies$ **strong LP**.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \leq y \leq 1$) of:

$$\min y \text{ s.t. } \quad x \geq 1, \qquad x \leq My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \geq 1$, $x \leq y \Rightarrow y \geq 1$.
- Feasible LP region: only $x = 1$, $y = 1$.
- LP relaxation = integer solution $\implies$ **strong LP**.

Case 2: Loose $M = 1000$.

- Constraints: $x \geq 1$, $x \leq 1000y$.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \leq y \leq 1$) of:

$$\min y \text{ s.t.} \quad x \geq 1, \qquad x \leq My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \geq 1$, $x \leq y \Rightarrow y \geq 1$.
- Feasible LP region: only $x = 1$, $y = 1$.
- LP relaxation = integer solution $\implies$ **strong LP**.

Case 2: Loose $M = 1000$.

- Constraints: $x \geq 1$, $x \leq 1000y$.
- LP can pick $y = 0.001$, $x = 1$.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \leq y \leq 1$) of:

$$\min y \text{ s.t.} \quad x \geq 1, \qquad x \leq My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \geq 1$, $x \leq y \Rightarrow y \geq 1$.
- Feasible LP region: only $x = 1$, $y = 1$.
- LP relaxation = integer solution $\implies$ **strong LP**.

Case 2: Loose $M = 1000$.

- Constraints: $x \geq 1$, $x \leq 1000y$.
- LP can pick $y = 0.001$, $x = 1$.
- Objective y becomes tiny $\implies$ large integrality gap.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \le y \le 1$) of:

$$\min y \text{ s.t.} \quad x \ge 1, \qquad x \le My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \ge 1$, $x \le y \Rightarrow y \ge 1$.
- Feasible LP region: only $x = 1$, $y = 1$.
- LP relaxation = integer solution $\implies$ **strong LP**.

Case 2: Loose $M = 1000$.

- Constraints: $x \ge 1$, $x \le 1000y$.
- LP can pick $y = 0.001$, $x = 1$.
- Objective y becomes tiny $\implies$ large integrality gap.
- Branch-and-bound must search many nodes.

Tiny Big-M Demo: Tight vs Loose M

Consider relaxation (i.e. $0 \leq y \leq 1$) of:

$$\min y \text{ s.t. } \quad x \geq 1, \qquad x \leq My, \qquad y \in \{0, 1\}.$$

Case 1: Tight $M = 1$.

- Constraints: $x \geq 1$, $x \leq y \Rightarrow y \geq 1$.
- Feasible LP region: only $x = 1$, $y = 1$.
- LP relaxation = integer solution $\implies$ **strong LP**.

Case 2: Loose $M = 1000$.

- Constraints: $x \geq 1$, $x \leq 1000y$.
- LP can pick $y = 0.001$, $x = 1$.
- Objective y becomes tiny $\implies$ large integrality gap.
- Branch-and-bound must search many nodes.

Takeaway:

- M must be just large enough to model the logic correctly.
- Too-large M weakens the LP relaxation and slows the solver.

Special Ordered Sets: SOS1 and SOS2

Solvers support **Special Ordered Sets** patterns:

- SOS1: At most one variable in the set can be non-zero.
- SOS2: At most two adjacent variables (in a specified order) can be non-zero.

Special Ordered Sets: SOS1 and SOS2

Solvers support **Special Ordered Sets** patterns:

SOS1: At most one variable in the set can be non-zero.

SOS2: At most two adjacent variables (in a specified order) can be non-zero.

Usage:

- SOS1: choose exactly one option / segment / pattern.
- SOS2: piecewise-linear functions with convex hull interpolation.

Special Ordered Sets: SOS1 and SOS2

Solvers support **Special Ordered Sets** patterns:

SOS1: At most one variable in the set can be non-zero.

SOS2: At most two adjacent variables (in a specified order) can be non-zero.

Usage:

- SOS1: choose exactly one option / segment / pattern.
- SOS2: piecewise-linear functions with convex hull interpolation.

Benefit:

1. Solver can internally use clever heuristics designed for either SOS1/SOS2.

Gurobi: Using SOS1 for Choice Modeling

Example: choose exactly one option

```
import gurobipy as gp
from gurobipy import GRB
m = gp.Model("inventory_complementarity")
demand = 50
# Continuous decision variables
produced = m.addVar(lb=0, ub=supply, name="produced")
leftover = m.addVar(lb=0, name="leftover_stock")
backorder = m.addVar(lb=0, name="unmet_demand")
# produced - leftover + backorder = demand
m.addConstr(produced - leftover + backorder == demand)
# Complementarity: cannot have leftover AND backorder
m.addSOS(GRB.SOS_TYPE1, [leftover, backorder]) #New
# Penalties
m.setObjective(1*leftover + 100*backorder, GRB.MINIMIZE)
m.optimize()
```

Interpretation:

- SOS1 enforces mutual exclusivity. Exactly one of leftover or demand is non-zero.

Motivating SOS2: What Is a Piecewise-Linear Function?

A **piecewise-linear (PWL)** function is made of straight segments between known **breakpoints**:

$$(x_0, c_0), (x_1, c_1), \ldots, (x_K, c_K).$$

Motivating SOS2: What Is a Piecewise-Linear Function?

A **piecewise-linear (PWL)** function is made of straight segments between known **breakpoints**:

$$(x_0, c_0), (x_1, c_1), \ldots, (x_K, c_K).$$

Example:

k	0	1	2	3
x_k	0	50	100	200
c_k	0	40	150	320

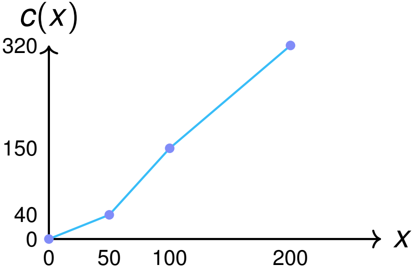
Motivating SOS2: What Is a Piecewise-Linear Function?

A **piecewise-linear (PWL)** function is made of straight segments between known **breakpoints**:

$$(x_0, c_0), (x_1, c_1), \ldots, (x_K, c_K).$$

Example:

k	0	1	2	3
x_k	0	50	100	200
c_k	0	40	150	320



Representing a Point on the Curve

We want to represent an arbitrary point (x, c) on this broken line.

Representing a Point on the Curve

We want to represent an arbitrary point (x, c) on this broken line.
Idea: write it as a **convex combination** of the breakpoints.

$$(x, c) = \sum_{k=0}^{K} \lambda_k (x_k, c_k), \qquad \sum_{k=0}^{K} \lambda_k = 1, \quad \lambda_k \geq 0.$$

Representing a Point on the Curve

We want to represent an arbitrary point (x, c) on this broken line.
Idea: write it as a **convex combination** of the breakpoints.

$$(x, c) = \sum_{k=0}^{K} \lambda_k (x_k, c_k), \qquad \sum_{k=0}^{K} \lambda_k = 1, \quad \lambda_k \geq 0.$$

Example: Suppose the true point is halfway between $(x_1, c_1) = (50, 40)$ and $(x_2, c_2) = (100, 150)$.

Representing a Point on the Curve

We want to represent an arbitrary point (x, c) on this broken line.
Idea: write it as a **convex combination** of the breakpoints.

$$(x, c) = \sum_{k=0}^{K} \lambda_k (x_k, c_k), \qquad \sum_{k=0}^{K} \lambda_k = 1, \quad \lambda_k \geq 0.$$

Example: Suppose the true point is halfway between $(x_1, c_1) = (50, 40)$ and $(x_2, c_2) = (100, 150)$.

$$\lambda_1 = 0.5, \quad \lambda_2 = 0.5, \quad \text{others} = 0.$$

Then

$$x = 50(0.5) + 100(0.5) = 75, \qquad c = 40(0.5) + 150(0.5) = 95.$$

$(x, c) = (75, 95)$ lies exactly on the line segment between breakpoints 1 and 2.

Why We Need an Additional Rule

The convex combination equations alone allow mixtures of *non-adjacent* breakpoints:

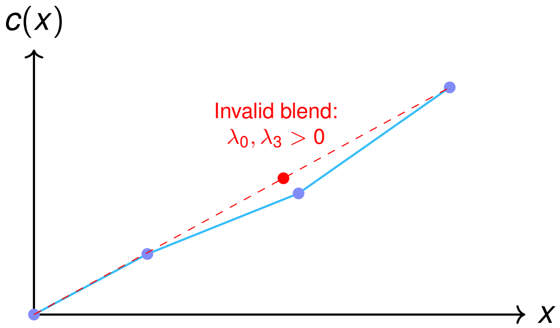
$$\lambda_0 = 0.3,\ \lambda_3 = 0.7 \ \Rightarrow\ x = 140,\ c = 224.$$

Why We Need an Additional Rule

The convex combination equations alone allow mixtures of *non-adjacent* breakpoints:

$$\lambda_0 = 0.3,\ \lambda_3 = 0.7 \ \Rightarrow\ x = 140,\ c = 224.$$

But that point is **not on the curve**—it “cuts across” segments.



We must ensure that at most **two adjacent** λ_k are positive.

Special Ordered Sets Type 2 (SOS2)

The adjacency rule is enforced by declaring the $(\lambda_0, \ldots, \lambda_K)$ as an **SOS2 set** ordered by x_k.

Special Ordered Sets Type 2 (SOS2)

The adjacency rule is enforced by declaring the $(\lambda_0, \ldots, \lambda_K)$ as an **SOS2 set** ordered by x_k.

SOS2 definition:

- At most two λ_k can be nonzero, and **adjacent in the order of** x_k.

Special Ordered Sets Type 2 (SOS2)

The adjacency rule is enforced by declaring the $(\lambda_0, \ldots, \lambda_K)$ as an **SOS2 set** ordered by x_k.

SOS2 definition:

- At most two λ_k can be nonzero, and **adjacent in the order of** x_k.

Gurobi: Declaring an SOS2 Set

Example code for a piecewise-linear cost:

```
import gurobipy as gp
from gurobipy import GRB

xs = [0, 50, 100, 200]
cs = [0, 40, 150, 320]   # cost at breakpoints

m = gp.Model("piecewise_cost")

lam = m.addVars(len(xs), lb=0.0, name="lam")
x   = m.addVar(lb=0.0, name="x")
c   = m.addVar(lb=0.0, name="cost")

# Convex combination for x
m.addConstr(gp.quicksum(lam[k] for k in range(len(xs))) == 1)
m.addConstr(x == gp.quicksum(xs[k] * lam[k] for k in range(len(xs))))

# SOS2 declaration: at most two adjacent lambdas > 0
m.addSOS(GRB.SOS_TYPE2, [lam[k] for k in range(len(xs))], xs) #New!!

# Final convex combination for cost
m.addConstr(c == gp.quicksum(cs[k] * lam[k] for k in range(len(xs))))
```

Homework: Building SOS1 and SOS2 from Scratch

In this week's HW, you'll **implement the same ideas manually** using big-M and binary variables. For SOS2:

Homework: Building SOS1 and SOS2 from Scratch

In this week's HW, you'll **implement the same ideas manually** using big-M and binary variables. For SOS2:
Step 1: Start from the breakpoints

$$(x_k, c_k) = (0, 0), (50, 40), (100, 150), (200, 320)$$

and define binary variables $z_k \in \{0, 1\}$ that pick which segment is active.

Homework: Building SOS1 and SOS2 from Scratch

In this week's HW, you'll **implement the same ideas manually** using big-M and binary variables. For SOS2:
Step 1: Start from the breakpoints

$$(x_k, c_k) = (0,0), (50,40), (100,150), (200,320)$$

and define binary variables $z_k \in \{0,1\}$ that pick which segment is active.
Step 2: Use Big-M or indicator constraints to:

- enforce that exactly one segment is active,
- interpolate x and c correctly within that segment.

Goal: understand how SOS1/SOS2 encapsulates the same logic you'd otherwise build with Big-M + binaries.

Summary of Lecture 6

- We saw core **binary modeling patterns**:
  - selection / knapsack,
  - fixed-charge on/off and linking constraints,
  - logical implications and precedence,
  - either–or disjunctions via binaries.
- We discussed **Big-M**:
  - too-large $M \Rightarrow$ weak LP, big B&B tree,
  - use data/context to tighten M and strengthen the formulation.
- We introduced **SOS1/SOS2**:
  - solver-native constructs for at-most-one and piecewise-linear modeling,
  - avoid manual Big-M and get stronger relaxations.