← Back to Home

🏭 ERP.AI Engineering Assessment

Part 2: Factory Steady State & Bounded Belts

📋 Overview

Build **two command-line tools** that **read JSON on stdin** and **write JSON on stdout**. No extra prints or logs. Deterministic results for identical inputs.

CLI Contract & Performance

```
factory < input.json > output.json
belts   < input.json > output.json
```

⚡ **Each case must complete in ≤ 2 seconds on a normal laptop.**

🏭 A) Factory Steady State (with cycles)

Goal

Given:

- **Machines** with base speed
- **Recipes** with inputs, outputs, and craft time
- Optional **modules** per machine type (speed, productivity)
- **Limits** on raw inputs and on machine counts
- A **target item** with a required production rate (items/min)

Compute non-negative **crafts per minute** `x_r` for each recipe so that:

1. The **target item** is produced at the requested rate (exact match).
2. All **intermediates** are perfectly balanced (steady state).
3. **Raw items** are net-consumed only and never exceed their supply caps.
4. **Machine caps** per type are respected.

If multiple solutions exist, choose one that uses the **fewest total machines**. If infeasible, return the **maximum feasible target rate** and brief bottleneck hints.

Units

`items/min` for item flows; `crafts/min` for recipe completions.

Effective speed with modules

Modules apply per machine type (not per-recipe). All recipes run on the same machine type inherit the same speed and productivity modifiers.

For recipe `r` running on machine type `m`:

- Base machine speed: `machines[m].crafts_per_min`
- Module speed multiplier: `1 + modules[m].speed` (default `0` if no module)
- Recipe time: `time_s(r)` seconds

Effective crafts/min for `r`:

```
eff_crafts_per_min(r) = machines[m].crafts_per_min * (1 + speed) * 60 / time_s(r)
```

Productivity

Productivity multiplies **outputs only** (not inputs). It applies uniformly to all output items of the recipe.

If recipe `r` has `out_r = { item_A: 2, item_B: 1 }` and module on machine `m` has `prod = 0.2`:

- Effective outputs per craft: `{ item_A: 2 * 1.2, item_B: 1 * 1.2 }`

Conservation (steady state)

Let `x_r ≥ 0` be crafts/min for recipe `r`. For each item `i`:

```
∑_r [out_r[i] * (1 + prod_m) * x_r] - ∑_r [in_r[i] * x_r] = b[i]
```

Where:

- For the **target** item `t`: `b[t] = target_rate` (exact production required)
- For **intermediates** (items produced by some recipe and consumed by others, excluding raw and target): `b[i] = 0` (perfect balance)
- For **raw items** with supply caps: `b[i] ≤ 0` (net consumption) and `|b[i]| ≤ raw_cap[i]`

Cycles & Byproducts: Recipes may form cycles (A→B→A) or produce byproducts. The steady-state conservation equations handle these naturally—set `b[i] = 0` for all intermediates, including cyclic ones. If a recipe produces more than it consumes, accumulate the surplus; if it consumes more, draw from elsewhere (or from raw supply).

Machine usage constraints

For each recipe `r` on machine `m`:

```
machines_used_by_r = x_r / eff_crafts_per_min(r)
```

Per-type cap:

```
∑_{r uses m} machines_used_by_r ≤ max_machines[m]
```

Objective (Tie-break)

Minimize total machines used:

```
minimize: ∑_m ∑_{r uses m} x_r / eff_crafts_per_min(r)
```

When multiple feasible solutions exist (e.g., alternative recipes), prefer the one using fewer total machines. This is a **secondary objective** after feasibility; use a two-phase approach:

1. **Phase 1**: Find any feasible solution satisfying conservation, caps, and target rate.
2. **Phase 2**: Reoptimize within the feasible region to minimize total machines.

Alternatively, formulate as a single linear program with the objective weighted appropriately.

Output schema

Success

```
{
  "status": "ok",
  "per_recipe_crafts_per_min": {
    "recipeA": 123.0,
    "recipeB": 0.0
  },
  "per_machine_counts": {
    "assembler_1": 10.5,
    "chemical": 8.0
  },
  "raw_consumption_per_min": {
    "iron_ore": 240.0,
    "copper_ore": 360.0
  }
}
```

Infeasible

```
{
  "status": "infeasible",
  "max_feasible_target_per_min": 980.0,
  "bottleneck_hint": [
    "assembler_1 cap",
    "iron_ore supply"
  ]
}
```

Sample (input → output)

Input

```
{
  "machines": {
    "assembler_1": {"crafts_per_min": 30},
    "chemical": {"crafts_per_min": 60}
  },
  "recipes": {
    "iron_plate": {
      "machine": "chemical",
      "time_s": 3.2,
      "in": {"iron_ore": 1},
      "out": {"iron_plate": 1}
    },
    "copper_plate": {
      "machine": "chemical",
      "time_s": 3.2,
      "in": {"copper_ore": 1},
      "out": {"copper_plate": 1}
    },
    "green_circuit": {
      "machine": "assembler_1",
      "time_s": 0.5,
      "in": {"iron_plate": 1, "copper_plate": 3},
      "out": {"green_circuit": 1}
    }
  },
  "modules": {
    "assembler_1": {"prod": 0.1, "speed": 0.15},
    "chemical": {"prod": 0.2, "speed": 0.1}
  },
  "limits": {
    "raw_supply_per_min": {"iron_ore": 5000, "copper_ore": 5000},
    "max_machines": {"assembler_1": 300, "chemical": 300}
  },
  "target": {"item": "green_circuit", "rate_per_min": 1800}
}
```

Output

```
{
  "status": "ok",
  "per_recipe_crafts_per_min": {
    "iron_plate": 1800.0,
    "copper_plate": 5400.0,
    "green_circuit": 1800.0
  },
  "per_machine_counts": {
    "chemical": 50.0,
    "assembler_1": 60.0
  },
  "raw_consumption_per_min": {
    "iron_ore": 1800.0,
    "copper_ore": 5400.0
  }
}
```

Numeric Tolerance

- Conservation equations: `|balance[i]| < 1e-9` (absolute tolerance).
- Raw consumption feasibility: `consumption ≤ cap + 1e-9`.
- Machine usage: `usage ≤ max + 1e-9`.
- Determinism: Use consistent floating-point arithmetic; if using a solver, seed it for reproducibility.

Acceptance & Determinism

- Exact conservation within tight tolerances (no "almost zero" leaks).
- Raw consumption ≤ caps; machine counts ≤ caps.
- If multiple solutions exist with identical machine count: output is deterministic (e.g., tie-break by recipe name lexicographically).
- Output is deterministic given identical input.

🔀 B) Belts with Bounds and Node Caps

Goal

On a directed conveyor graph:

- Each edge `(u → v)` has **lower** `lo` and **upper** `hi` flow bounds (items/min).
- Certain nodes have **throughput caps** on total in or out.
- Multiple **sources** with fixed supplies; a single **sink** with demand equal to total supply.

Decide feasibility while respecting all bounds and caps; if feasible, produce one valid flow. If infeasible, provide a clear certificate.

Concepts

- **Edge flow:** `lo ≤ f_e ≤ hi`.
- **Node conservation:** inflow + supply = outflow + demand (only sink has demand).
- **Node caps:** enforce via node-splitting or constraint.
- **Sources:** provide fixed supply (no incoming edges).
- **Sink:** single node receiving all flow (no outgoing edges).

Method (high level)

1. Node Capacity Handling

For each capped node `v` (not source or sink):

- Split into two nodes: `v_in` (receives all incoming) and `v_out` (sends all outgoing).
- Add edge `v_in → v_out` with capacity `cap(v)`.
- Redirect all incoming edges to `v_in`, all outgoing to `v_out`.

2. Lower Bounds via Transformation

For each edge `(u → v)` with `lo, hi`:

- Reduce the edge capacity to `hi - lo`.
- Accumulate imbalance: `+lo` at `v` (demand), `-lo` at `u` (supply).
- After solving, add `lo` back to the computed flow to recover original values.

3. Feasibility Check (Lower Bounds)

- Create a super-source `s*` and super-sink `t*`.
- For each node with positive imbalance (demand), connect `s*` to it with that demand.
- For each node with negative imbalance (supply), connect it to `t*` with that supply.
- Run max-flow from `s*` to `t*` with adjusted capacities.
- If max-flow saturates all required edges, lower bounds are **feasible**.

4. Main Flow (Supplies to Sink)

- Connect each source to the actual sink with its supply.
- Run max-flow from source side to sink.
- Reconstruct original flows by adding `lo` back on each edge.

Output schema

Success

```
{
  "status": "ok",
  "max_flow_per_min": 1500,
  "flows": [
    {"from": "s1", "to": "a", "flow": 900},
    {"from": "a", "to": "b", "flow": 900},
    {"from": "b", "to": "sink", "flow": 900},
    {"from": "s2", "to": "a", "flow": 600},
    {"from": "a", "to": "c", "flow": 600},
    {"from": "c", "to": "sink", "flow": 600}
  ]
}
```

Infeasible

```
{
  "status": "infeasible",
  "cut_reachable": ["s1", "a", "b"],
  "deficit": {
    "demand_balance": 300,
    "tight_nodes": ["b"],
    "tight_edges": [
      {
        "from": "b",
        "to": "sink",
        "flow_needed": 300
      }
    ]
  }
}
```

- `cut_reachable`: Nodes reachable from source in the residual graph when flow is maximal (defines the bottleneck).
- `demand_balance`: Unsatisfied demand on the source side of the cut.
- `tight_nodes`: Nodes at capacity limits within the cut.
- `tight_edges`: Edges at capacity limits crossing the cut (from reachable to unreachable).

Numeric Tolerance

- Edge flow: `lo ≤ f ≤ hi + 1e-9`.
- Node conservation: `|imbalance| < 1e-9`.
- Determinism: Tie-break augmenting paths lexicographically or use consistent solver configuration.

📦 Submission Package

Organize your repo as:

```
part2_assignment/
├─ factory/
│  └─ main.py              # executable: reads JSON from stdin, writes JSON to stdout
├─ belts/
│  └─ main.py              # executable: reads JSON from stdin, writes JSON to stdout
├─ tests/
│  ├─ test_factory.py
│  └─ test_belts.py
├─ verify_factory.py       # (optional) validation helper
├─ verify_belts.py         # (optional) validation helper
├─ gen_factory.py          # (optional) test case generator
├─ gen_belts.py            # (optional) test case generator
├─ run_samples.py
├─ README.md               # design note (1–2 pages)
└─ RUN.md                  # exact run commands
```

Rules

- Read JSON from **stdin**; write a **single JSON object** to **stdout**.
- **No** extra prints, logs, or debug output.
- **Deterministic** output for identical inputs.
- **≤ 2 seconds** per case.

README.md must cover

Factory modeling choices:

- How item balances and conservation equations are enforced.
- Raw consumption and machine capacity constraints.
- Module application (per-machine-type).
- Handling of cycles, byproducts, and self-contained recipes.
- How ties in machine count are broken.
- Infeasibility detection and reporting (binary search for max rate vs. linear program relaxation?).

Belts modeling choices:

- Max-flow with lower bounds: transformation steps and order of operations.
- Node-splitting for capacity constraints.
- Feasibility check strategy.
- How infeasibility certificates (min-cut) are computed and reported.

Numeric approach:

- Tolerances used (`1e-9`, etc.).
- Linear programming solver or hand-rolled algorithm (and why).
- Tie-breaking strategy for determinism.

Failure modes & edge cases:

- Cycles in recipes.
- Infeasible raw supplies or machine counts.
- Degenerate or redundant recipes.
- Disconnected graph components (belts).

RUN.md example

```
# Run sample tests
python run_samples.py "python factory/main.py" "python belts/main.py"

# Run pytest
FACTORY_CMD="python factory/main.py" BELTS_CMD="python belts/main.py" pytest -q
```

Deliverables checklist

- ✅ `factory` and `belts` directories with `main.py` executable.
- ✅ Provided tests pass locally.
- ✅ JSON output matches schemas exactly.
- ✅ No extraneous output (only final JSON).
- ✅ Deterministic results.
- ✅ ≤ 2 seconds per test case.
- ✅ README and RUN.md complete.

Packaging

1. Create a GitHub repository named `factorio-assignment-<yourname>` with `public` access.
2. Place top-level folder at repo root: `part2_assignment/`
3. Include a top-level `SUBMISSION.txt` with:
   - Name, email
   - OS, CPU, RAM
   - Language & version (e.g., Python 3.11.7)
   - Any optional notes (e.g., solver libraries, performance notes)
4. Submit the GitHub URL for your repository.

🚀 Submit Your Solution

Create a GitHub repository with the structure above and submit the URL below:

https://github.com/your-username/factorio-assignment-yourname

🎯 Submit Repository

⚠️ *Expect a 20-30 minute defense call where you'll explain your modeling choices and algorithm decisions.*